Automatic Analysis of Pointer Aliasing for Untyped Programs ⋆

A. Venet [1]

LIX, École Polytechnique, 91128 Palaiseau, France.
venet@lix.polytechnique.fr
http://lix.polytechnique.fr/~venet

In this paper we describe an automatic analysis based on Abstract Interpretation that discovers potential sharing relationships among the data structures created by an imperative program. The analysis is able to distinguish between elements in inductively defined structures and does not require any explicit data type declaration by the programmer. In order to construct the abstract interpretation we introduce a new class of abstract domains: the *cofibered domains*.

Key words: alias analysis, storeless semantics, abstract interpretation, cofibered domain, widening operator, finite-state automaton, abstract numerical domain.

1 Introduction

Alias analysis consists of automatically inferring an approximate but sound description of pointer equality relationships during the execution of a program. This problem turns out to be particularly difficult when we allow structured data to be dynamically allocated on the heap and modified via destructive updating. Deutsch [Deu92b,Deu94] has designed a very accurate analysis based on the framework of Abstract Interpretation [CC77,CC92a] which can be applied to such programs, and which is able to distinguish between elements of recursively defined data structures. In that analysis no particular assumptions are made on the program except one: there must be explicit data type declarations describing the shape of the structures to which each variable in the program may point, the analysis algorithm strongly relying on that piece of

⋆ Expanded version of a paper presented at the third Static Analysis Symposium, Aachen, Germany, 1996 [Ven96].

[1] Partially supported by ESPRIT BRA 8130 LOMAPS.

information. In this paper we propose to remove this restriction by designing an alias analysis which could be applied to untyped or dynamically typed programs without any further assumption, while still ensuring a comparable level of accuracy.

The major difficulty lies in the impossibility of splitting up the structural and aliasing information. The interdependence between the two is due to the presence of destructive assignment. For instance, consider the instruction `x.f := y` written in an imperative language with mutable records. Its effect is to assign the value of the pointer `y` to the field `f` of the record pointed to by `x`. Performing a descriptive data type inference (using a grammar-based analysis [Hei94,HJ94,CC95b] for example) necessarily requires information about all possible aliases of `x` in order to propagate the structural modification induced by the assignment. Conversely, performing an alias analysis necessarily requires a description of the data structures, the information being expressed as aliasing relationships between access paths in these very structures. Since we do not make any assumptions on the data structures and aliasing relationships created by a program during its execution, we have to perfom both analyses *simultaneously*.

The core of this work consists of constructing an abstract domain for describing aliasing relationships amongst elements of recursively defined data structures which are not statically known. The idea is to represent sets of access paths within data structures by finite-state automata, whereas alias pairs are described by means of numerical constraints on the number of times each transition of an automaton may be used. We can then describe nonuniform aliasing relations such as: "two lists of arbitrary lengths may only share elements lying at the same rank". Hence, we have to combine an infinite collection of abstract domains of numerical constraints describing aliasing relationships, each of these domains being parameterized by a finite-state automaton describing some data structure. This construction characterizes a new class of abstract domains, the *cofibered domains* [Ven96], which enjoy nice compositional properties, in particular for the design of *widening operators* [CC92a,CC92b].

The techniques required to reach this level of expressivity are not elementary as one could expect. Therefore, we will try to keep the presentation as simple as possible, sacrificing accuracy and efficiency matters to clarity whenever necessary. As a basis for our analysis we use a simple imperative language with mutable records and dynamic memory allocation, which is described in Sect. 2. The language is given a *storeless* semantics, i.e. the memory is not represented as a graph but as the set of access paths in the data structures together with an aliasing relation on these paths. In Sect. 3 we define the abstract interpretation framework that we will use to specify the analysis. Section 4 presents the techniques of cofibered domains. This is applied in Sect. 5 to design the abstract cofibered domain of data types and aliasing

relationships. In order to make the abstract interpretation computable, we need to build widening operators over this domain that ensure the termination of the analysis. This is the purpose of Sect. 6. Finally the abstract semantics of the language is described in Sect. 7.

2 Storeless Semantics of a Simple Untyped Language

Following [Cou81], we use a simple language in which programs are described by *dynamic systems* to illustrate our analysis. More precisely, a program P is given by a set $\mathfrak{P}$ of *program points*, an *entry point* $\mathfrak{e} \in \mathfrak{P}$, a set $\mathfrak{T} \subseteq \mathfrak{P}$ of *terminal points* and a *transition relation* $\longrightarrow$ between program points. Given two program points $\mathfrak{p}$ and $\mathfrak{p}'$, a transition $\mathfrak{p} \xrightarrow{\mathfrak{i}} \mathfrak{p}'$ is labelled by an instruction $\mathfrak{i}$. The program operates on a set $\mathfrak{V} = \{x_1, \ldots, x_m\}$ of variables that point to data structures built upon a signature $\mathfrak{R} = \{\mathfrak{r}_1, \ldots, \mathfrak{r}_n\}$ of records. For each record $\mathfrak{r} \in \mathfrak{R}$, we denote by $\mathfrak{F}(\mathfrak{r}) = \{\mathfrak{f}_1, \ldots, \mathfrak{f}_k\}$ the set of its fields. A record $\mathfrak{r}$ such that $\mathfrak{F}(\mathfrak{r}) = \emptyset$ is called an *atom*. Note that there is no conditional statement in our language but mere nondeterminism. This choice leads to a very simple language which allows us to concentrate uniquely on pointer aliasing problems. The set $\mathfrak{I}$ of instructions is defined as follows:

$$
\begin{array}{rll}
\mathfrak{I} ::= & x := \mathbf{new}\ \mathfrak{r} & \text{record allocation} \\
\mid & x := y & \text{variable aliasing} \\
\mid & x := y.\mathfrak{r}\sharp\mathfrak{f} & \text{pointer assignment} \\
\mid & x.\mathfrak{r}\sharp\mathfrak{f} := y & \text{destructive assignment}
\end{array}
$$

where $\mathfrak{r} \in \mathfrak{R}$, $\mathfrak{f} \in \mathfrak{F}(\mathfrak{r})$ and x, y are *distinct* variables of $\mathfrak{V}$. The condition on the variables simplifies the definition of the semantics but is in no way restrictive, since any assignment instruction can be reduced to this form by introducing auxiliary variables. A program that creates two lists of pairwise aliased elements is given graphically in Figure 1. The set $\mathfrak{V}$ of program variables is $\{x, y, x', y', z\}$. The signature is made of two atoms `nil` and `symb`, and a record `cons` with two fields: `car` and `cdr`.

Classically we would model a memory configuration by using an *environment* ρ and a *store* $\mathfrak{S}$. The store is a labelled graph (V, λ, E) where the vertices V are *memory locations*. The labelling function $\lambda : V \longrightarrow \mathfrak{R}$ maps each location to the record it contains, each edge $v \xrightarrow{\mathfrak{f}} v'$ of E being labelled by a field $\mathfrak{f} \in \mathfrak{F}(\lambda(v))$. The environment is a function $\rho : \mathfrak{V} \longrightarrow V$ that maps each variable of the program to a location in the store. Jonkers [Jon81] pointed out that such a model is too coarse since it has to take *garbage collection* into account explicitly. Indeed, locations which are unreachable from the variables

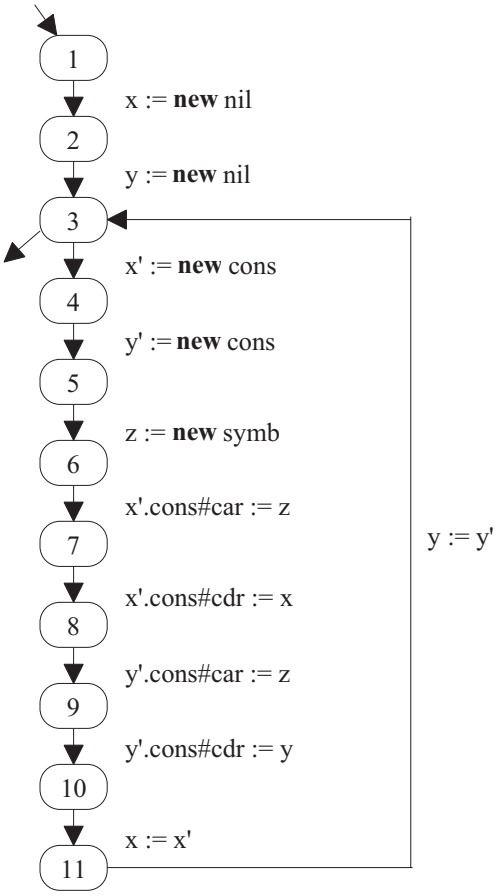


Figure 1. Sample program

of the program may be present in the store. Therefore, Jonkers advocated for a storeless representation of the memory made of the set Π of all access paths in the data structures starting from the variables of the program, together with an equivalence relation $\equiv$, the *aliasing relation*, on Π. The aliasing relation equates access paths that lead to the same location. All reachable locations of the original store can thus be retrieved from the quotient set $\Pi/_{\equiv}$. This semantic model is particularly well-suited to our problem, since it expresses all information about data structures and aliasing in a minimal and canonical way. It has been successfully used by Deutsch [Deu92b,Deu94] in the design of his alias analysis.

We now define this model more formally. Let $\mathfrak{M}$ be the set of all memory configurations. An element $\mathfrak{m}$ of $\mathfrak{M}$ is a pair $(\Pi, \equiv)$ where:

- Π is a *prefix-closed* subset of $\mathfrak{V}.\Sigma^* + \varepsilon$, where $\Sigma = \bigcup_{\mathfrak{r}\in\mathfrak{R}}\{\mathfrak{r}\sharp\mathfrak{f} \mid \mathfrak{f} \in \mathfrak{F}(\mathfrak{r})\}$ is the set of all data selectors.
- $\equiv$ is an equivalence relation on Π, the *aliasing relation*, such that:

- $\equiv$ is *right-regular*:

$$\forall \pi, \pi' \in \Pi : \forall \sigma \in \Sigma : (\pi \equiv \pi' \wedge \pi.\sigma \in \Pi) \Longrightarrow (\pi'.\sigma \in \Pi \wedge \pi.\sigma \equiv \pi'.\sigma)$$

- $\forall \pi \in \Pi : \varepsilon \equiv \pi \Longrightarrow \pi = \varepsilon$

The prefix-closedness condition expresses that Π is a *tree domain*, which corresponds to the complete unfolding of the store in the classical model. The empty path ε represents the origin of all access paths, i.e. the whole memory, and is therefore not aliased to any other path. The right-regularity of the aliasing relation $\equiv$ simply means that whenever two access paths π_1 and π_2 are aliased, i.e. point to the same data structure, then all common descendants $\pi_1.\pi$ and $\pi_2.\pi$ of these paths within the shared structure are also aliased. Note that we do not represent atomic values in this semantic model in order to keep the structure of memory configurations simple.

We define the set $\mathfrak{C}$ of semantic configurations of the program as $(\mathfrak{P} \times \mathfrak{M}) \cup \{\Omega\}$, where Ω is a special configuration denoting a runtime error. The semantics of an instruction $\mathfrak{i}$ is given by a transformer of memory configurations $[\![\mathfrak{i}]\!] : \mathfrak{M}_\mathfrak{i} \longrightarrow \mathfrak{M}$, where $\mathfrak{M}_\mathfrak{i} \subseteq \mathfrak{M}$ is the set of memory configurations to which it makes sense to apply instruction $\mathfrak{i}$. Then, the semantics of the program is defined by a transition system $(\mathfrak{C}, \longrightarrow)$ as follows:

$$\frac{\mathfrak{p} \xrightarrow{\mathfrak{i}} \mathfrak{p}' \wedge \mathfrak{m} \in \mathfrak{M}_\mathfrak{i}}{(\mathfrak{p}, \mathfrak{m}) \longrightarrow (\mathfrak{p}', [\![\mathfrak{i}]\!]\mathfrak{m})} \qquad \frac{\mathfrak{p} \xrightarrow{\mathfrak{i}} \mathfrak{p}' \wedge \mathfrak{m} \notin \mathfrak{M}_\mathfrak{i}}{(\mathfrak{p}, \mathfrak{m}) \longrightarrow \Omega}$$

The initial semantic configuration $\mathfrak{c}_0$ is $(\mathfrak{e}, (\varepsilon + \mathfrak{V}, \{(\varepsilon, \varepsilon)\} \cup \{(x, x) \mid x \in \mathfrak{V}\}))$, since at the beginning of the execution of a program, no data structure is allocated in memory and all variables are uninitialized. It now remains to define precisely the semantics $[\![\mathfrak{i}]\!]$ of an instruction $\mathfrak{i}$.

The set $\mathfrak{M}$ can be ordered by the componentwise inclusion which we denote by $\subseteq_\mathfrak{M}$. It is readily checked that the intersection of any family of aliasing relations is still an aliasing relation. Moreover, the pair $\top_\mathfrak{M} = (\mathfrak{V}.\Sigma^* + \varepsilon, \mathfrak{V}.\Sigma^* \times \mathfrak{V}.\Sigma^* \cup \{(\varepsilon, \varepsilon)\})$ is trivially a memory configuration and, for any $\mathfrak{m} \in \mathfrak{M}$, we have $\mathfrak{m} \subseteq_\mathfrak{M} \top_\mathfrak{M}$. Therefore, the poset $(\mathfrak{M}, \subseteq_\mathfrak{M})$ can be endowed with the structure of a complete lattice $(\mathfrak{M}, \subseteq_\mathfrak{M}, \bot_\mathfrak{M}, \cup_\mathfrak{M}, \top_\mathfrak{M}, \cap_\mathfrak{M})$, the meet operation $\cap_\mathfrak{M}$ corresponding to componentwise intersection. Given any prefix-closed set $\Pi \subseteq \mathfrak{V}.\Sigma^* + \varepsilon$ of access paths and any binary relation ρ on $\Pi \setminus \{\varepsilon\}$, we denote by $\wp_\mathfrak{M}(\Pi, \rho)$ the memory configuration $\bigcap_\mathfrak{M}\{\mathfrak{m} \in \mathfrak{M} \mid (\Pi, \rho) \subseteq_\mathfrak{M} \mathfrak{m}\}$. Thus, $\wp_\mathfrak{M}$ can be seen as a closure operation producing a valid memory configuration from a partial specification of access paths and aliasing relationships.

Now, let $\mathfrak{m} = (\Pi, \equiv)$ be a memory configuration. The semantics of an allocation instruction $[\![x := \mathbf{new}\, \mathfrak{r}]\!]$ is defined for every memory configuration $\mathfrak{m} = (\Pi, \equiv)$ for which $x \in \Pi$, and maps $\mathfrak{m}$ to $(\Pi', \equiv')$, where:

– $\Pi' = (\Pi \setminus x.\Sigma^+) \cup \{x.\mathfrak{r}\sharp\mathfrak{f} \mid \mathfrak{f} \in \mathfrak{F}(\mathfrak{r})\}$
– $\equiv' = (\equiv \cap (\Pi \setminus x.\Sigma^*) \times (\Pi \setminus x.\Sigma^*)) \cup \{(x, x)\}$

The effect of the allocation instruction is to remove all access paths starting from x, as well as all related alias pairs, and to add the access paths corresponding to the fields of record $\mathfrak{r}$. The newly created access paths are of course unaliased.

The semantics of all other assignment instructions can be expressed with a single operation $\mathbf{set}(\pi_1.\sigma, \pi_2)$ on $\mathfrak{M}$, where π_1, π_2 are access paths and $\sigma \in \mathfrak{V} \cup \Sigma$. This operation is defined for every memory configuration $\mathfrak{m} = (\Pi, \equiv)$ satisfying the following conditions:

(i) $\{\pi_1.\sigma, \pi_2\} \subseteq \Pi$
(ii) $\pi_2 \notin [\pi_1]_\equiv.\sigma.\Sigma^*$

where $[\pi_1]_\equiv$ denotes the equivalence class of π_1 modulo $\equiv$. The memory configuration $\mathbf{set}(\pi_1.\sigma, \pi_2)\mathfrak{m}$ is given by $\wp_{\mathfrak{M}}(\Pi', \rho)$, where:

– $\Pi' = (\Pi \setminus [\pi_1]_\equiv.\sigma.\Sigma^*) \cup \{\pi_1.\sigma\}$
– $\rho = (\equiv \cap (\Pi \setminus [\pi_1]_\equiv.\sigma.\Sigma^*) \times (\Pi \setminus [\pi_1]_\equiv.\sigma.\Sigma^*)) \cup \{(\pi_1.\sigma, \pi_2)\}$

Note that this definition does make sense, since we easily check that the set Π' is prefix-closed. This operation amounts to performing a destructive assignment in memory at the location pointed to by $\pi_1.\sigma$. All other paths accessing to the same location and below, i.e. those lying in $[\pi_1]_\equiv.\sigma.\Sigma^*$, are removed from the configuration, as well as all related alias pairs. The aliasing relationship $\pi_1.\sigma \equiv \pi_2$ is then enforced. The semantics of assignment instructions can thus be given as follows:

$$[\![x := y]\!]\mathfrak{m} = \mathbf{set}(x, y)\mathfrak{m}$$
$$[\![x := y.\mathfrak{r}\sharp\mathfrak{f}]\!]\mathfrak{m} = \mathbf{set}(x, y.\mathfrak{r}\sharp\mathfrak{f})\mathfrak{m}$$
$$[\![x.\mathfrak{r}\sharp\mathfrak{f} := y]\!]\mathfrak{m} = \mathbf{set}(x.\mathfrak{r}\sharp\mathfrak{f}, y)\mathfrak{m}$$

The set $\mathfrak{M}_\mathfrak{i}$ of memory configurations upon which the semantics of an assignment instruction $\mathfrak{i}$ is defined, is that of the corresponding $\mathbf{set}$ operation. Note that a runtime error only occurs whenever condition (i) is not met, which corresponds to accessing to unallocated memory. Since by definition, $x \neq y$ and $[\varepsilon]_\equiv = \{\varepsilon\}$, condition (ii) is always satisfied in the above cases.

Our purpose is to infer automatically an approximate description of the memory configurations at every point of the program during its execution. We are therefore interested in the *collecting semantics* [Cou81] $\mathcal{S} = \{\mathfrak{c} \mid \mathfrak{c}_0 \xrightarrow{*} \mathfrak{c}\}$ of the program, which is the set of all configurations that can be derived from the initial one with respect to the operational semantics.

Example 1 *If we consider the program represented in Figure 1, every memory configuration* $\mathfrak{m}$ *such that* $(3, \mathfrak{m}) \in \mathcal{S}$*, is given by* $\varrho_{\mathfrak{M}}(\Pi, \rho)$*, where:*

$$\begin{cases} \Pi = \{x, y, x', y'\}.(\{\texttt{cons}\sharp\texttt{cdr}^n\} \cup \{\texttt{cons}\sharp\texttt{cdr}^i.\texttt{cons}\sharp\texttt{car} \mid 0 \leq i < n\}) \cup \{z\} \\ \rho = \{(x.\texttt{cons}\sharp\texttt{cdr}^i.\texttt{cons}\sharp\texttt{car}, y.\texttt{cons}\sharp\texttt{cdr}^i.\texttt{cons}\sharp\texttt{car}) \mid 0 \leq i < n\} \\ \qquad \cup \{(x, x'), (y, y'), (x.\texttt{cons}\sharp\texttt{car}, z)\} \quad \textit{if } n \geqslant 1 \end{cases}$$

for a certain integer $n \geqslant 0$ *denoting the number of times the body of the loop has been executed. This means that at the end of the program,* x *and* y *point to two lists of the same length* n *with pairwise aliased elements.*

We will design a computable approximation of the collecting semantics of a program by using the techniques of Abstract Interpretation.

3 Abstract Interpretation

Abstract Interpretation [CC77,CC79,Cou81,CC92a] is a theory which provides a number of general frameworks for defining approximations of semantic specifications. A *semantic specification* [CC92a] is typically given by a poset $(\mathcal{D}, \sqsubseteq)$, the *semantic domain* of concrete properties, an endomorphism F of $(\mathcal{D}, \sqsubseteq)$, the *semantic function*, and an element $\bot$ of $\mathcal{D}$, the *basis*, such that the least fixpoint $\mathrm{lfp}_\bot \mathrm{F}$ of F greater than or equal to $\bot$ exists. The latter condition is usually ensured by the fact that $\mathcal{D}$ is a complete lattice and $\bot$ is a prefixpoint, i.e. $\bot \sqsubseteq \mathrm{F}(\bot)$. This is what happens in our case, where the semantic domain $\mathcal{D}$ associated to a program is given by the set $\wp((\mathfrak{P} \times \mathfrak{M}) \cup \{\Omega\})$ ordered by inclusion. F is the $\cup$-complete function that maps any element $d \in \mathcal{D}$ to $\{\mathfrak{c}_0\} \cup \{\mathfrak{c}' \mid \exists \mathfrak{c} \in d : \mathfrak{c} \longrightarrow \mathfrak{c}'\}$ and $\bot$ is the empty set. A standard result [CC77,Cou81] states that the collecting semantics $\mathcal{S}$ is equal to $\mathrm{lfp}_\bot \mathrm{F}$. Hence, by Kleene's theorem $\mathcal{S}$ is given by the limit of the following iteration sequence:

$$\begin{cases} \mathrm{F}_0 = \bot \\ \mathrm{F}_{n+1} = \mathrm{F}(\mathrm{F}_n) \end{cases}$$

i.e. the least upper bound af all iterates. In general this sequence is not ultimately stationary and neither is its limit finitely representable. We are therefore led to define a *semantic approximation* of $\mathcal{S}$ which is more abstract, in the sense that it does not capture all properties expressed by $\mathcal{S}$, but which is on the other hand computable.

A semantic approximation is formally given by an *abstract semantic specification*. There are several ways to construct an abstract semantic specification, the most well-known being the one based on *semi-dual Galois connec-*

tions [CC77,CC79]. However, this model cannot be applied to our problem because we will use regular abstractions of sets of access paths, and in general there exists no best approximation of an arbitrary set of strings by a regular language (see [CC95b] for more details). We will use instead a relaxed framework [CC92a,CC92b] defined as follows. An abstract semantic specification is given by a preordered set $(\mathcal{D}^\sharp, \preceq)$, the *abstract semantic domain*, related to $\mathcal{D}$ by a *concretization function* $\gamma : \mathcal{D}^\sharp \longrightarrow \mathcal{D}$, an *abstract basis* $\bot^\sharp \in \mathcal{D}^\sharp$ and an *abstract semantic function* $\mathrm{F}^\sharp : \mathcal{D}^\sharp \longrightarrow \mathcal{D}^\sharp$, such that the following soundness conditions are met:

- $\bot \sqsubseteq \gamma(\bot^\sharp)$
- $\forall d_1^\sharp, d_2^\sharp \in \mathcal{D}^\sharp : d_1^\sharp \preceq d_2^\sharp \Longrightarrow \gamma(d_1^\sharp) \sqsubseteq \gamma(d_2^\sharp)$
- $\forall d^\sharp \in \mathcal{D}^\sharp : \mathrm{F} \circ \gamma(d^\sharp) \sqsubseteq \gamma \circ \mathrm{F}^\sharp(d^\sharp)$

The abstract semantics $\mathcal{S}^\sharp$ is obtained by mimicking the calculation of $\mathcal{S}$ as the limit of an iteration sequence. In order to define this abstract iteration sequence and to guarantee its convergence on a sound approximation of $\mathcal{S}$, we need a *widening operator* [Cou81,CC92a] $\nabla : \mathcal{D}^\sharp \times \mathcal{D}^\sharp \longrightarrow \mathcal{D}^\sharp$ satisfying the following properties:

- For all $d_1^\sharp, d_2^\sharp$ in $\mathcal{D}^\sharp$, $d_1^\sharp \preceq d_1^\sharp \nabla d_2^\sharp$ and $d_2^\sharp \preceq d_1^\sharp \nabla d_2^\sharp$.
- For every sequence $(d_n^\sharp)_{n \geqslant 0}$ of elements of $\mathcal{D}^\sharp$, the sequence $(d_n^\nabla)_{n \geqslant 0}$ inductively defined as follows:

$$\begin{cases} d_0^\nabla &= d_0^\sharp \\ d_{n+1}^\nabla &= d_n^\nabla \nabla d_{n+1}^\sharp \end{cases}$$

  is ultimately stationary.

The *abstract iteration sequence with widening* $(\mathrm{F}_n^\nabla)_{n \geqslant 0}$ is thus inductively defined as:

$$\begin{cases} \mathrm{F}_0^\nabla &= \bot^\sharp & \\ \mathrm{F}_{n+1}^\nabla &= \mathrm{F}_n^\nabla & \text{if } \mathrm{F}^\sharp(\mathrm{F}_n^\nabla) \preceq \mathrm{F}_n^\nabla \\ &= \mathrm{F}_n^\nabla \nabla \mathrm{F}^\sharp(\mathrm{F}_n^\nabla) & \text{otherwise} \end{cases}$$

Theorem 2 (Abstract iteration [CC92a,CC92b]) *The abstract iteration sequence with widening* $(\mathrm{F}_n^\nabla)_{n \geqslant 0}$ *is ultimately stationary and its limit* $\mathcal{S}^\sharp$ *satisfies* $\mathcal{S} \sqsubseteq \gamma(\mathcal{S}^\sharp)$. *Furthermore, if* N *is an integer such that* $\mathrm{F}_N^\nabla = \mathrm{F}_{N+1}^\nabla$, *then* $\forall n \geqslant N : \mathrm{F}_n^\nabla = \mathrm{F}_N^\nabla = \mathcal{S}^\sharp$.

We can clearly derive a semantic analyzer from this theorem. The rest of this paper will be entirely devoted to the construction of the abstract semantic specification of a program.

Since we are mainly interested in describing access paths and aliasing information, we first abstract $\mathcal{D}$ in order to forget about runtime errors. Let $\mathcal{D}^\sharp_\Omega$ be the set $\wp(\mathfrak{P} \times \mathfrak{M})$ ordered by inclusion. The concretization function $\gamma_\Omega : (\mathcal{D}^\sharp_\Omega, \subseteq) \longrightarrow (\mathcal{D}, \subseteq)$ maps any set C of configurations to $C \cup \{\Omega\}$. Now, suppose that we are provided with a preordered set $(\mathfrak{M}^\sharp, \preceq^{\mathfrak{M}})$ of abstract memory configurations related to $\wp(\mathfrak{M})$ via a concretization map $\gamma^{\mathfrak{M}} : (\mathfrak{M}^\sharp, \preceq^{\mathfrak{M}}) \longrightarrow (\wp(\mathfrak{M}), \subseteq)$. Given a set X and a finite collection I of indices, we denote by $\prod_{i \in I} X$ the set of I-indexed families of elements of X, which we will equally view as functions from I into X. Then, we construct $\mathcal{D}^\sharp$ as $\prod_{\mathfrak{p} \in \mathfrak{P}} \mathfrak{M}^\sharp$, the preorder $\preceq$ being the pointwise extension of $\preceq^{\mathfrak{M}}$. We also introduce a concretization function $\gamma' : (\mathcal{D}^\sharp, \preceq) \longrightarrow (\mathcal{D}^\sharp_\Omega, \subseteq)$ which maps any $C^\sharp \in \mathcal{D}^\sharp$ to the set $\{(\mathfrak{p}, \mathfrak{m}) \mid \mathfrak{m} \in \gamma^{\mathfrak{M}}(C^\sharp(\mathfrak{p}))\}$. The global concretization function $\gamma : (\mathcal{D}^\sharp, \preceq^{\mathfrak{M}}) \longrightarrow (\mathcal{D}, \subseteq)$ is then given by the compound map $\gamma_\Omega \circ \gamma'$. We will design the domain $\mathfrak{M}^\sharp$ of abstract memory configurations stepwise by successive approximations of $\wp(\mathfrak{M})$.

A key step in the construction of $\mathfrak{M}^\sharp$ consists of collapsing a set of memory configurations into a singleton. More precisely, let $\mathfrak{M}^\sharp_0$ be the collection of all pairs (Π, ρ), where Π is a prefix-closed subset of $\mathfrak{V}.\Sigma^* + \varepsilon$ and ρ is a mere binary relation on Π. We denote by $\mathrm{D}(\Pi)$ the *diagonal* of Π, that is the set $\{(\pi, \pi) \mid \pi \in \Pi\}$. The concretization function $\gamma^{\mathfrak{M}}_0 : \mathfrak{M}^\sharp_0 \longrightarrow \wp(\mathfrak{M})$ maps any $(\Pi, \rho) \in \mathfrak{M}^\sharp_0$ to $\{(\Pi', \equiv) \in \mathfrak{M} \mid \Pi' \subseteq \Pi \wedge \equiv \subseteq \rho \cup \mathrm{D}(\Pi)\}$. We do not encode reflexivity in ρ since this information is entirely determined by Π. The preorder $\preceq^{\mathfrak{M}}_0$ on $\mathfrak{M}^\sharp_0$ is given by pointwise set inclusion, thus making the function $\gamma^{\mathfrak{M}}_0$ monotone.

Then, we make data type information explicit by using deterministic finite-state automata to represent sets of access paths. A deterministic finite-state automaton $\mathcal{A}$ is given by a finite set Q of *states*, an *initial state* $i \in Q$ and a collection $T \subseteq Q \times \Sigma \times Q$ of *transitions* labelled by data selectors, such that for any $q \in Q$ and any $\sigma \in \Sigma$, there exists at most one transition $q \xrightarrow{\sigma} q'$ in T. Since we only cope with prefix-closed sets of access paths, all states of an automaton are terminal. For foundational reasons we suppose that the states of all automata come from a single infinite set $\mathcal{Q}$. We denote by $\mathcal{L}(\mathcal{A})$ the language recognized by $\mathcal{A}$. Since we only consider deterministic automata, there is a one-to-one correspondence between paths in an automaton $\mathcal{A}$ and access paths in $\mathcal{L}(\mathcal{A})$. A *morphism* $f : \mathcal{A}_1 \longrightarrow \mathcal{A}_2$ between $\mathcal{A}_1 = (Q_1, i_1, T_1)$ and $\mathcal{A}_2 = (Q_2, i_2, T_2)$ is a function $f : Q_1 \longrightarrow Q_2$ such that the following conditions are satisfied:

(i) $f(i_1) = i_2$
(ii) $\forall (q_1, \sigma, q_2) \in T_1 : (f(q_1), \sigma, f(q_2)) \in T_2$

Deterministic finite-state automata together with associated morphisms clearly form a category **Aut**.

The existence of a morphism from $\mathcal{A}_1$ to $\mathcal{A}_2$ trivially implies that $\mathcal{L}(\mathcal{A}_1) \subseteq \mathcal{L}(\mathcal{A}_2)$. Thus, morphisms provide us with a way of comparing the *structure* of automata. They will play a major role in the design of the abstract domain $\mathfrak{M}^\sharp$, as we will see in Sect. 5. Now, let $\mathfrak{M}_1^\sharp$ be the collection of all pairs $(\mathbf{A}, \rho)$ such that $\mathbf{A} \in \prod_{x \in \mathfrak{V}} \mathbf{Aut}$ and ρ is a binary relation on $\bigcup_{x \in \mathfrak{V}} x.\mathcal{L}(\mathbf{A}(x))$. Then, we put $(\mathbf{A}_1, \rho_1) \preceq_1^{\mathfrak{M}} (\mathbf{A}_2, \rho_2)$ whenever there exist morphisms $f_x : \mathbf{A}_1(x) \longrightarrow \mathbf{A}_2(x)$ for every $x \in \mathfrak{V}$, and $\rho_1 \subseteq \rho_2$. We relate $\mathfrak{M}_1^\sharp$ to $\mathfrak{M}_0^\sharp$ via a concretization function $\gamma_1^{\mathfrak{M}} : \mathfrak{M}_1^\sharp \longrightarrow \mathfrak{M}_0^\sharp$ which maps any $(\mathbf{A}, \rho) \in \mathfrak{M}_1^\sharp$ to $(\bigcup_{x \in \mathfrak{V}} x.\mathcal{L}(\mathbf{A}(x)) \cup \{\varepsilon\}, \rho \cup \{(\varepsilon, \varepsilon)\})$. This function is clearly monotone with respect to $\preceq_1^{\mathfrak{M}}$ and $\preceq_0^{\mathfrak{M}}$. For simplicity we have removed the empty path ε from the abstract memory configuration, since it is never involved in aliasing relationships but trivial ones.

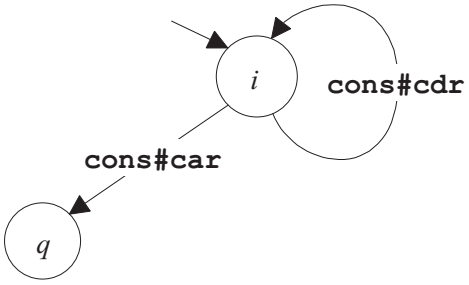
An abstract aliasing relation ρ over a set of access paths described by a tuple $\mathbf{A}$ of deterministic finite-state automata is in general not finitely representable. Therefore, we have to design a computable approximation of ρ which is able to capture non-uniform relationships between elements of recursively defined data structures. The idea introduced by Deutsch [Deu92a,Deu92b] consisted of using Eilenberg's unitary-prefix monomial decomposition of a regular language [Eil74] to represent sets of access paths. In this framework an abstract path is given by $\sigma_0.B_0^*.\sigma_1 \ldots \sigma_n.B_n^*.\sigma_{n+1}$ where the σ_i are data selectors and the B_i are regular languages, called the *bases* of the decomposition. The bases characterize the recursive parts of data structures. An abstract aliasing relationship is then represented by a pair of such paths together with numerical constraints on the number of iterations of each basis. Using this domain with linear equality constraints [Kar76], we can describe *exactly* the aliasing relation of Example 1. However, Deutsch's analysis strongly relies on data type information provided by the programmer and there is no simple way of extending it to the untyped case.

We use a similar abstraction which will moreover allow us to infer the aliasing relation in the same time as the description of access paths. The idea is to abstract a path in an automaton by a tuple of integers representing the number of times each transition of the automaton occurs in the path. Therefore, if $\mathcal{A} = (Q, i, T)$ is a deterministic finite-state automaton, we associate a counter $q.\sigma$ to each transition $q \xrightarrow{\sigma} q'$ in T, and we denote by $\mathrm{C}(\mathcal{A})$ the set of all these counters. For any $q \in Q$, let $\mathcal{L}_q(\mathcal{A})$ be the set of words of Σ^* labelling a path from i to q in $\mathcal{A}$. For every $\pi \in \mathcal{L}(\mathcal{A})$ and any $c \in \mathrm{C}(\mathcal{A})$, we denote by $||\pi||_c$ the number of times the transition corresponding to the counter c occurs in the path of $\mathcal{A}$ which is labelled by π. Note that this definition does make sense because, $\mathcal{A}$ being deterministic, the path labelled by π is unique.

Now, let $\mathbf{A} \in \prod_{x \in \mathfrak{V}} \mathbf{Aut}$ be a tuple of automata where, for any $x \in \mathfrak{V}$, $\mathbf{A}(x) = (Q_x, i_x, T_x)$. We denote by $\hat{\mathbf{A}}$ the set of pairs $\varkappa = \langle (x, q), (y, q') \rangle$ where $x, y \in \mathfrak{V}$, $q \in Q_x$ and $q' \in Q_y$. We denote by $\mathrm{C}(\varkappa)$ the set of counters $\mathbf{l}.c$ and

$\mathbf{r}.c'$, where $c \in \mathrm{C}(\mathbf{A}(x))$ and $c' \in \mathrm{C}(\mathbf{A}(y))$. Symbols $\mathbf{l}$ and $\mathbf{r}$ respectively stand for "left" and "right", and are used to distinguish between transition counters of the automata associated to the left and right components of an abstract alias pair $\langle (x,q),(y,q')\rangle$. Note that we cannot just simply use the variable names to make the distinction, because we may have $x = y$. Let ρ be a binary relation on $\bigcup_{x\in\mathfrak{V}} x.\mathcal{L}(\mathbf{A}(x))$ and $r = (x.\pi_1, y.\pi_2) \in \rho$, where $\pi_1 \in \mathcal{L}_q(\mathbf{A}(x))$ and $\pi_2 \in \mathcal{L}_{q'}(\mathbf{A}(y))$. We put $\varkappa(r) = \langle (x,q),(y,q')\rangle$ and we denote by $\hat{r}$ the element of $\prod_{k\in\mathrm{C}(\varkappa(r))} \mathbb{N}$ which maps any $\mathbf{l}.c$ to $\|\pi_1\|_c$ and any $\mathbf{r}.c'$ to $\|\pi_2\|_{c'}$. Then, we define $\mathfrak{M}_2^\sharp$ to be the set of pairs $(\mathbf{A},\mathbf{R})$, where $\mathbf{R}$ is an element of $\prod_{\varkappa\in\hat{\mathbf{A}}} \wp(\prod_{k\in\mathrm{C}(\varkappa)} \mathbb{N})$. The concretization function $\gamma_2^{\mathfrak{M}} : \mathfrak{M}_2^\sharp \longrightarrow \mathfrak{M}_1^\sharp$ maps any $(\mathbf{A},\mathbf{R})$ to $(\mathbf{A},\rho)$, where ρ is the set of all pairs $r = (x.\pi_1, y.\pi_2)$ such that $\hat{r} \in \mathbf{R}(\varkappa(r))$.

Example 3 *Take* $\mathfrak{V} = \{x,y\}$ *and* $\mathfrak{R} = \{\mathtt{cons},\mathtt{nil}\}$, *with* $\mathfrak{F}(\mathtt{cons}) = \{\mathtt{car},\ \mathtt{cdr}\}$ *and* $\mathfrak{F}(\mathtt{nil}) = \emptyset$. *Let* $\mathcal{A}$ *be the following automaton:*



Let $(\mathbf{A},\mathbf{R})$ *be an element of* $\mathfrak{M}_2^\sharp$ *such that* $\mathbf{A}(x) = \mathbf{A}(y) = \mathcal{A}$ *and, putting* $\varkappa = \langle (x,q),(y,q)\rangle$,

$$\mathbf{R}(\varkappa) = \{\nu \in \prod_{k\in\mathrm{C}(\varkappa)} \mathbb{N} \mid \nu(\mathbf{l}.i.\mathtt{cons}\sharp\mathtt{cdr}) = \nu(\mathbf{r}.i.\mathtt{cons}\sharp\mathtt{cdr})\}$$

The description of access paths shows that x *and* y *point to mere list structures, whereas the aliasing relation says that both lists may only share elements lying at the same rank.*

Difficulties arise when it comes to defining the approximation preorder $\preceq_2^{\mathfrak{M}}$ on $\mathfrak{M}_2^\sharp$. Intuitively, we would say that $(\mathbf{A}_1,\mathbf{R}_1) \preceq_2^{\mathfrak{M}} (\mathbf{A}_2,\mathbf{R}_2)$ whenever there are morphisms $f_x : \mathbf{A}_1(x) \longrightarrow \mathbf{A}_2(x)$ for every $x \in \mathfrak{V}$, such that the "image" of $\mathbf{R}_1$ by these morphisms is safely approximated by $\mathbf{R}_2$. Indeed, $\mathbf{R}_1$ is described by using the transition counters of the automata in $\mathbf{A}_1$, which means that we first have to "transfer" the aliasing relation $\mathbf{R}_1$ into the counting domain defined by the automata in $\mathbf{A}_2$ in order to compare it with $\mathbf{R}_2$. This approximation structure on $\mathfrak{M}_2^\sharp$ is characteristic of a general class of abstract domains, the *cofibered domains*, which we will now introduce.

4 Cofibered Domains

Intuitively, a cofibered domain is the result of "glueing" a collection of base posets, each of these giving an abstract description of a same concrete domain. The "glue" which allows us to link these posets together is defined by a categorical structure. More precisely, the existence of a morphism $f : P_1 \longrightarrow P_2$ between posets P_1 and P_2 means that P_2 is an approximation of P_1. Furthermore, the morphism provides us with a consistent way of expressing any abstract value of P_1 into the coarser domain P_2. Hence, at any point of the computation an abstract value always belongs to some base poset. A cofibered domain can thus be seen as a *dynamic poset*, for the approximation structure is allowed to change during the computation of the abstract iteration sequence, transitions between different posets being carried through via morphisms.

We now give a formal construction of cofibered domains. We denote by **Cat** the category of small categories with functors, by **Proset** the category of preordered sets with monotone maps, and by **Poset** the category of partially ordered sets with monotone maps. Let $\Delta : \mathbf{C} \longrightarrow \mathbf{Poset}$ be a functor from a small category $\mathbf{C}$ into **Poset.** For each object A of $\mathbf{C}$, we call the poset ΔA the *fiber* of Δ over A, and we denote by $\leq_A$ the order relation on ΔA. The *Grothendieck construction* [BW90] associates to Δ the category $\mathbf{G}\Delta$ defined as follows:

(i) An object of $\mathbf{G}\Delta$ is a pair (A, x) where x is an element of ΔA.
(ii) A morphism $f : (A, x) \longrightarrow (B, y)$ is given by a morphism $f : A \longrightarrow B$ in $\mathbf{C}$ such that $\Delta f(x) \leq_B y$. The composition of morphisms in $\mathbf{G}\Delta$ is described in Figure 2.

Let $\mathbf{U} : \mathbf{Cat} \longrightarrow \mathbf{Proset}$ be the forgetful functor [Mac71] that sends every category to its underlying preordered set obtained by collapsing each Hom-set into one arrow.

Definition 4 *A* cofibered domain *is a preordered set $P = (E, \preceq)$ such that there exists a functor $\Delta : \mathbf{C} \longrightarrow \mathbf{Poset}$ verifying $P = \mathbf{UG}\Delta$. The functor Δ is called the* display *of the domain and $\mathbf{C}$ its* base.

The denomination "cofibered domain" comes from the fact that $\mathbf{G}\Delta$ can be turned into a cofibration over $\mathbf{C}$. The Grothendieck construction is actually a canonical way of building cofibrations [BW90]. In the following we will frequently identify a cofibered domain with its display.

For example, the domains $\mathfrak{M}_0^\sharp$ and $\mathfrak{M}_1^\sharp$ introduced in Sect. 3 are cofibered**.** The base category $\mathbf{C}_0$ of $\mathfrak{M}_0^\sharp$ is given by the collection of all prefix-closed subsets of $\mathfrak{V}.\Sigma^* + \varepsilon$, whereas the arrows of $\mathbf{C}_0$ are merely the inclusion relations $\Pi_1 \subseteq \Pi_2$. The display $\Delta_0 : \mathbf{C}_0 \longrightarrow \mathbf{Poset}$ maps every Π in $\mathbf{C}_0$ to the poset

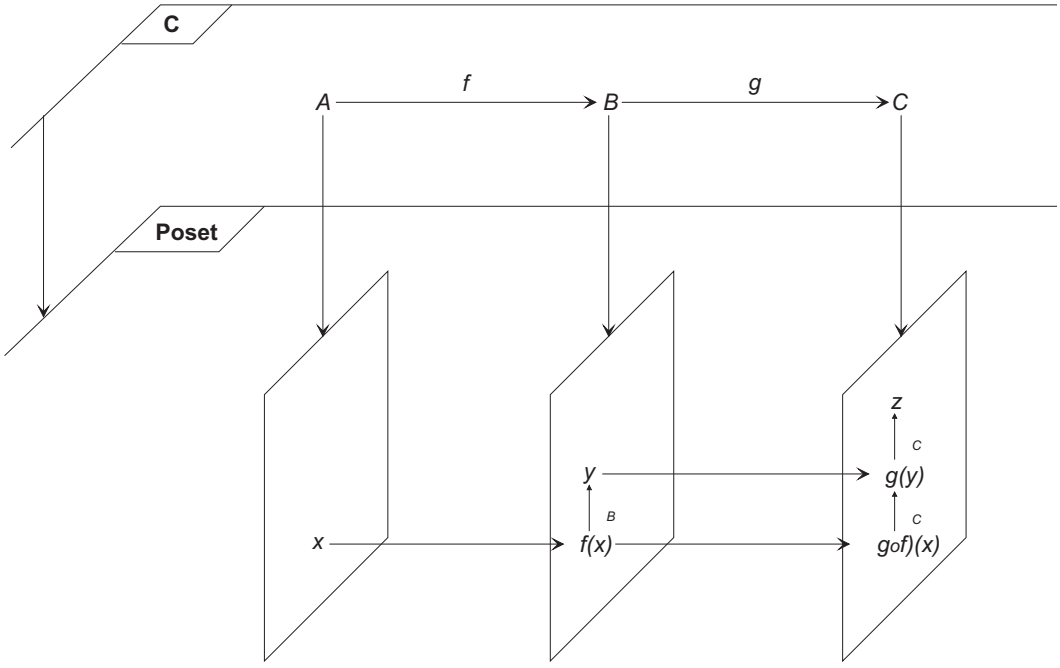
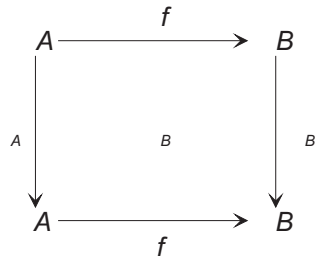


Figure 2. Composition of morphisms in the Grothendieck construction

$(\wp(\Pi \times \Pi), \subseteq)$. The image of an arrow $\Pi_1 \subseteq \Pi_2$ is the canonical inclusion map from $\wp(\Pi_1 \times \Pi_1)$ into $\wp(\Pi_2 \times \Pi_2)$. Then, we readily check that $(\mathfrak{M}_0^\sharp, \preceq_0^{\mathfrak{M}}) = \mathbf{UG}\Delta_0$. The base category $\mathbf{C}_1$ of $\mathfrak{M}_1^\sharp$ is simply the product category $\prod_{x \in \mathfrak{V}} \mathbf{Aut}$. The display $\Delta_1 : \mathbf{C}_1 \longrightarrow \mathbf{Poset}$ maps any tuple $\mathbf{A}$ of automata to the poset $(\wp(\bigcup_{x \in \mathfrak{V}} x.\mathcal{L}(\mathbf{A}(x)) \times \bigcup_{x \in \mathfrak{V}} x.\mathcal{L}(\mathbf{A}(x))), \subseteq)$. The image of a tuple $\mathbf{f} : \mathbf{A}_1 \longrightarrow \mathbf{A}_2$ of morphisms of automata is given by the canonical inclusion map from $\Delta_1\mathbf{A}_1$ into $\Delta_1\mathbf{A}_2$. We easily check that $(\mathfrak{M}_1^\sharp, \preceq_1^{\mathfrak{M}}) = \mathbf{UG}\Delta_1$.

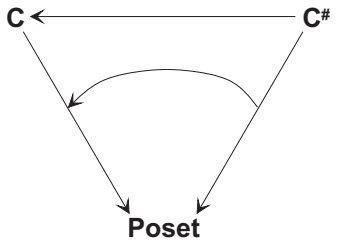
Cofibered domains enjoy the important property of *compositionality*: an approximation $\gamma : \mathbf{UG}\Delta^\sharp \longrightarrow \mathbf{UG}\Delta$ can be constructed piecewise from approximations of the components of the domain $\mathbf{UG}\Delta$, i.e. from both its base and fibers. The piecewise approximation of cofibered domains requires a relaxed notion of natural transformation between poset-valued functors.

Definition 5 (Lax natural transformation [Kel74]) *A* lax natural transformation $\kappa : \Delta^\sharp \longrightarrow \Delta$ *between two functors* $\Delta, \Delta^\sharp : \mathbf{C} \longrightarrow \mathbf{Poset}$ *is a collection of morphisms* $\kappa_A : \Delta^\sharp A \longrightarrow \Delta A$ *for each object* A *of* $\mathbf{C}$ *such that, for every morphism* $f : A \longrightarrow B$ *in* $\mathbf{C}$*, the following diagram lax commutes:*



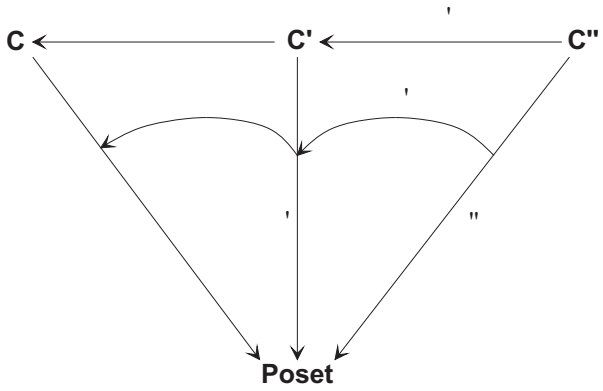
This can be equivalently reformulated as: $\Delta f \circ \kappa_A \leq_B \kappa_B \circ \Delta^\sharp f$.

A *fiberwise approximation* of a cofibered domain $\Delta : \mathbf{C} \longrightarrow \mathbf{Poset}$ is given by an abstract domain $\Delta^\sharp : \mathbf{C}^\sharp \longrightarrow \mathbf{Poset}$ together with a functor $\Gamma : \mathbf{C}^\sharp \longrightarrow \mathbf{C}$ and a lax natural transformation $\kappa : \Delta^\sharp \longrightarrow \Delta\Gamma$. This is summarized by the following diagram:



Γ is a concretization functor which approximates the base of the domain. The lax natural transformation is a collection of concretization functions between the abstract fibers of $\Delta^\sharp$ and the concrete ones of Δ, whence the name of fiberwise approximation. The lax commuting diagrams of Definition 5 express that every morphism $\Delta^\sharp f$ between abstract fibers is a *sound approximation* of the corresponding morphism Δf between concrete ones.

Fiberwise approximation defines a notion of morphism between cofibered domains. Following [Kel74], it can be shown that cofibered domains with such morphisms form a category that we denote by **CFD**. The composition of two morphisms:



is given by *diagram pasting* [KS74]. It is the morphism $(\overline{\Gamma}, \overline{\kappa}) : \Delta'' \longrightarrow \Delta$ defined as $\overline{\Gamma} = \Gamma\Gamma'$ and $\overline{\kappa}_A = \kappa_{\Gamma' A} \circ \kappa'_A$ for every object A of $\mathbf{C}''$. The Grothendieck construction induces a functor $\mathbf{G} : \mathbf{CFD} \longrightarrow \mathbf{Cat}$ [Kel74]. The image of a morphism $(\Gamma, \kappa) : \Delta^\sharp \longrightarrow \Delta$ is the functor $\mathbf{G}(\Gamma, \kappa) : \mathbf{G}\Delta^\sharp \longrightarrow \mathbf{G}\Delta$ that maps every object (A, x) to $(\Gamma A, \kappa_A(x))$ and every morphism $f : (A, x) \longrightarrow (B, y)$ to $\Gamma f : (\Gamma A, \kappa_A(x)) \longrightarrow (\Gamma B, \kappa_B(y))$. Therefore, the functor $\mathbf{UG} : \mathbf{CFD} \longrightarrow \mathbf{Proset}$ maps fiberwise approximations of cofibered domains to approximations between the underlying preordered sets.

For example, the concretization function $\gamma_1^{\mathfrak{M}} : (\mathfrak{M}_1^\sharp, \preceq_1^{\mathfrak{M}}) \longrightarrow (\mathfrak{M}_0^\sharp, \preceq_0^{\mathfrak{M}})$ defined in Sect. 3 actually comes from a fiberwise approximation. Let $\Gamma_1 : \mathbf{C}_1 \longrightarrow \mathbf{C}_0$ be the functor that sends any tuple of automata $\mathbf{A}$ to the set $\bigcup_{x \in \mathfrak{V}} x.\mathcal{L}(\mathbf{A}(x)) \cup \{\varepsilon\}$ and every tuple of morphisms $\mathbf{f} : \mathbf{A}_1 \longrightarrow \mathbf{A}_2$ to the inclusion arrow $\Gamma_1\mathbf{A}_1 \subseteq \Gamma_1\mathbf{A}_2$. Now, let $\kappa^1 : \Delta_1 \longrightarrow \Delta_0\Gamma_1$ be the lax natural transformation given as follows: for any $\mathbf{A}$ in $\mathbf{C}_1$, the function $\kappa^1_{\mathbf{A}} : \Delta_1\mathbf{A} \longrightarrow \Delta_0\Gamma_1\mathbf{A}$ maps every $\rho \in \Delta_1\mathbf{A}$ to $\rho \cup \{(\varepsilon, \varepsilon)\}$. Then, we easily check that $\gamma_1^{\mathfrak{M}} = \mathbf{UG}(\Gamma_1, \kappa^1)$. We are now able to complete the construction of the abstract domain $(\mathfrak{M}_2^\sharp, \preceq_2^{\mathfrak{M}})$.

5 The Abstract Domain of Data Types and Aliasing Relations

At the end of Sect. 3 we noticed that the approximation structure on $\mathfrak{M}_2^\sharp$ should rely on a mechanism that would allow us to "transfer" abstract aliasing relations via morphisms of automata. This amounts to defining a method for transferring the numerical abstraction of an access path along a morphism of automata. Let $\mathcal{A}_1 = (Q_1, i_1, T_1)$ and $\mathcal{A}_2 = (Q_2, i_2, T_2)$ be two automata of $\mathbf{Aut}$ connected by a morphism $f : \mathcal{A}_1 \longrightarrow \mathcal{A}_2$. If $i_1 \xrightarrow{\sigma_1} q_1 \cdots q_{n-1} \xrightarrow{\sigma_n} q_n$ is a path in $\mathcal{A}_1$, then, by definition of morphisms of automata, $f(i_1) \xrightarrow{\sigma_1} f(q_1) \cdots f(q_{n-1}) \xrightarrow{\sigma_n} f(q_n)$ is also a path in $\mathcal{A}_2$. Moreover, this path is the unique one labelled by $\sigma_1 \ldots \sigma_n$ in $\mathcal{A}_2$, since all automata of $\mathbf{Aut}$ are deterministic. Therefore, if $\pi \in \mathcal{L}(\mathcal{A}_1) \subseteq \mathcal{L}(\mathcal{A}_2)$, then, for every $q_2.\sigma \in \mathrm{C}(\mathcal{A}_2)$, we have:

$$\|\pi\|_{q_2.\sigma} = \sum_{q_1 \in f^{-1}(q_2) \wedge q_1.\sigma \in \mathrm{C}(\mathcal{A}_1)} \|\pi\|_{q_1.\sigma}$$

Hence, we can transfer the numerical abstraction of access paths from $\mathcal{A}_1$ into $\mathcal{A}_2$ via the morphism f by means of elementary arithmetic operations. This simple observation will allow us to assign a cofibered structure to $\mathfrak{M}_2^\sharp$.

More precisely, let $\Delta_2 : \mathbf{C}_1 \longrightarrow \mathbf{Poset}$ be the functor which sends any tuple of automata $\mathbf{A}$ in $\mathbf{C}_1$ to the set $\prod_{\varkappa \in \hat{\mathbf{A}}} \wp(\prod_{k \in \mathrm{C}(\varkappa)} \mathbb{N})$ ordered by pointwise inclusion. Let $\mathbf{f} : \mathbf{A}_1 \longrightarrow \mathbf{A}_2$ be a morphism in $\mathbf{C}_1$. For any element $\varkappa_1 =$

$\langle (x, q_1), (y, q_1') \rangle$ of $\hat{\mathbf{A}}_1$, we denote by $\mathbf{f}(\varkappa_1)$ the element $\langle (x, \mathbf{f}(x)(q_1)), (y, \mathbf{f}(y)(q_1')) \rangle$ of $\hat{\mathbf{A}}_2$. Now, let $\tilde{\mathbf{f}}_{\varkappa_1} : \prod_{k \in \mathrm{C}(\varkappa_1)} \mathbb{N} \longrightarrow \prod_{k \in \mathrm{C}(\mathbf{f}(\varkappa_1))} \mathbb{N}$ be the function which maps any tuple of integers ν_1 to the tuple $\tilde{\mathbf{f}}_{\varkappa_1}(\nu_1)$, such that, for all $k \in \mathrm{C}(\mathbf{f}(\varkappa_1))$:

$$\tilde{\mathbf{f}}_{\varkappa_1}(\nu_1)(k) = \begin{cases} \sum_{q' \in \mathbf{f}(x)^{-1}(q) \wedge q'.\sigma \in \mathrm{C}(A(x))} \nu_1(\mathbf{l}.q.\sigma) & \text{if } k = \mathbf{l}.q.\sigma \\ \sum_{q' \in \mathbf{f}(y)^{-1}(q) \wedge q'.\sigma \in \mathrm{C}(A(y))} \nu_1(\mathbf{r}.q.\sigma) & \text{if } k = \mathbf{r}.q.\sigma \end{cases}$$

Then, the image of $\mathbf{f}$ by Δ_2 is the monotone map which sends any abstract aliasing relation $\mathbf{R} \in \Delta_2 \mathbf{A}_1$ to the relation $\Delta_2 \mathbf{f}(\mathbf{R})$ defined as follows:

$$\forall \varkappa_2 \in \hat{\mathbf{A}}_2 : \Delta_2 \mathbf{f}(\mathbf{R})(\varkappa_2) = \bigcup_{\mathbf{f}(\varkappa_1) = \varkappa_2} \{ \tilde{\mathbf{f}}_{\varkappa_1}(\nu_1) \mid \nu_1 \in \mathbf{R}(\varkappa_1) \}$$

We readily check that this definition does make sense (i.e. functoriality holds). Hence, we put $(\mathfrak{M}_2^\sharp, \preceq_2^{\mathfrak{M}}) = \mathbf{UG}\Delta_2$. The concretization function $\gamma_2^{\mathfrak{M}} : (\mathfrak{M}_2^\sharp, \preceq_2^{\mathfrak{M}}) \longrightarrow (\mathfrak{M}_1^\sharp, \preceq_1^{\mathfrak{M}})$ is given by $\mathbf{UG}(\mathrm{Id}_{\mathbf{C}_1}, \kappa^2)$, where $\mathrm{Id}_{\mathbf{C}_1}$ denotes the identity functor on $\mathbf{C}_1$ and, for any $\mathbf{A}$ in $\mathbf{C}_1$, the function $\kappa^2_{\mathbf{A}} : \Delta_2 \mathbf{A} \longrightarrow \Delta_1 \mathbf{A}$ maps every $\mathbf{R} \in \Delta_2 \mathbf{A}$ to the set of all pairs $r = (x.\pi_1, y.\pi_2)$ such that $\hat{r} \in \mathbf{R}(\varkappa(r))$. The monotonicity of $\kappa^2_{\mathbf{A}}$ and the lax-commutativity properties of κ^2 are straightforward. Note that all these definitions are consistent with the construction of $\mathfrak{M}_2^\sharp$ started in Sect. 3. Thus, the cofibered structure of $\mathfrak{M}_2^\sharp$ provides us with an approximation preorder which completely formalizes our first intuition of $\preceq_2^{\mathfrak{M}}$.

Abstract aliasing relations are still not representable since they may involve infinite sets of tuples of integers. Therefore, we must use a computable approximation of such sets, that is an *abstract numerical domain*. An abstract numerical domain is given by a collection of lattices $(\mathcal{N}_V, \sqsubseteq_V, \bot_V, \sqcup_V, \top_V, \sqcap_V)$ defined for any finite set V of variables. These lattices also come with concretization functions $\gamma_V : (\mathcal{N}_V, \sqsubseteq_V) \longrightarrow (\wp(\prod_{v \in V} \mathbb{N}), \subseteq)$. We will concentrate on *relational* domains, i.e. domains which are able to express linear constraints between variables. Several such domains have already been designed, each of those describing a particular kind of linear constraints : *equalities* [Kar76], *inequalities* [CH78] or *diophantine equations* [Gra91]. We leave the choice of $\mathcal{N}$ as a parameter of our analysis scheme and we refer the reader to the original papers for more details on the algorithmics of a particular domain. All the operations on $\mathcal{N}$ that we will need in the following will be defined independently of the abstract numerical domain. Nevertheless, we will frequently use Karr's domain of linear equalities [Kar76] to illustrate our constructions because of its very intuitive structure.

Example 6 *In Karr's domain of linear equalities an element N of $\mathcal{N}_V$ is*

the linear variety generated by a system $\{e_1, \ldots, e_n\}$ *of vectors of* $\prod_{v \in V} \mathbb{Q}$ *endowed with its canonical* $\mathbb{Q}$*-vector space structure, whereas* $\gamma_V(N)$ *is the set of integer-valued vectors lying in* N*.*

Now, let U, V and W be finite sets of variables. Every abstract numerical domain $\mathcal{N}$ associates to any *linear*[2] map $f : \prod_{v \in V} \mathbb{N} \longrightarrow \prod_{w \in W} \mathbb{N}$ a monotone function $\mathcal{N}f : (\mathcal{N}_V, \sqsubseteq_V) \longrightarrow (\mathcal{N}_W, \sqsubseteq_W)$ satisfying the following soundness condition:

$$\forall N \in \mathcal{N}_V : \forall \nu \in \gamma_V(N) : f(\nu) \in \gamma_W(\mathcal{N}f(N))$$

This abstraction preserves composition, that is, for any other linear map $g : \prod_{u \in U} \mathbb{N} \longrightarrow \prod_{v \in V} \mathbb{N}$, we have $\mathcal{N}(f \circ g) = \mathcal{N}f \circ \mathcal{N}g$, and the image of the identity function on $\prod_{v \in V} \mathbb{N}$ is the identity on $\mathcal{N}_V$. Moreover, the function $\mathcal{N}f$ is *additive*:

$$\forall N, N' \in \mathcal{N}_V : \mathcal{N}f(N \sqcup_V N') = \mathcal{N}f(N) \sqcup_W \mathcal{N}f(N')$$

Example 7 *We consider Karr's domain of linear equalities. Let* $\{e_1, \ldots, e_n\}$ *be a system of vectors generating* N*. If we denote by* $\bar{f}$ *the linear map* f *lifted to rational numbers, then* $\mathcal{N}f(N)$ *is the linear variety generated by the system* $\{\bar{f}(e_1), \ldots, \bar{f}(e_n)\}$*. If* N' *is another variety of* $\mathcal{N}_V$ *generated by* $\{f_1, \ldots, f_m\}$*, then* $N \sqcup_V N'$ *is the linear variety generated by* $\{e_1, \ldots, e_n, f_1, \ldots, f_m\}$*. Additivity of* $\mathcal{N}f$ *is quite obvious.*

We can then perform the last step in the construction of $(\mathfrak{M}^\sharp, \preceq^{\mathfrak{M}})$**.** We denote by $\Delta_3 : \mathbf{C}_1 \longrightarrow \mathbf{Poset}$ the functor which maps any tuple of automata $\mathbf{A}$ to the set $\prod_{\varkappa \in \hat{\mathbf{A}}} \mathcal{N}_{\mathrm{C}(\varkappa)}$ endowed with the product ordering**.** The image of a morphism $\mathbf{f} : \mathbf{A}_1 \longrightarrow \mathbf{A}_2$ is the monotone map which sends any $\mathbf{R}^\sharp \in \Delta_3\mathbf{A}_1$ to the abstract aliasing relation $\Delta_3\mathbf{f}(\mathbf{R}^\sharp)$ defined as follows:

$$\forall \varkappa_2 \in \hat{\mathbf{A}}_2 : \Delta_3\mathbf{f}(\mathbf{R}^\sharp)(\varkappa_2) = \bigsqcup\nolimits_{\mathrm{C}(\varkappa_2)} \{\mathcal{N}\tilde{\mathbf{f}}_{\varkappa_1}(\mathbf{R}^\sharp(\varkappa_1)) \mid \mathbf{f}(\varkappa_1) = \varkappa_2\}$$

Functoriality is ensured thanks to the additivity property of functions $\mathcal{N}\tilde{\mathbf{f}}_{\varkappa_1}$. Now, let $(\mathfrak{M}^\sharp_3, \preceq^{\mathfrak{M}}_3)$ be the cofibered domain $\mathbf{UG}\Delta_3$**.** We introduce a lax-natural transformation $\kappa^3 : \Delta_3 \longrightarrow \Delta_2$ such that, for each $\mathbf{A}$ in $\mathbf{C}_1$, the function $\kappa^3_{\mathbf{A}} : \Delta_3\mathbf{A} \longrightarrow \Delta_2\mathbf{A}$ maps any $\mathbf{R}^\sharp$ to the aliasing relation $\kappa^3_{\mathbf{A}}(\mathbf{R}^\sharp)$ defined as follows:

$$\forall \varkappa \in \hat{\mathbf{A}} : \kappa^3_{\mathbf{A}}(\mathbf{R}^\sharp)(\varkappa) = \gamma_{\mathrm{C}(\varkappa)}(\mathbf{R}^\sharp(\varkappa))$$

[2] The map f is *linear* if, for any $w \in W$, there exists $\alpha_w \in \prod_{v \in V} \mathbb{N}$ and $\beta_w \in \mathbb{N}$ such that, for all $\nu \in \prod_{v \in V} \mathbb{N}$, $f(\nu)(w) = \beta_w + \sum_{v \in V} \alpha_w(v)\nu(v)$.

Monotonicity and the lax-commutativity conditions are easily checked. We thus obtain a concretization function $\gamma_3^{\mathfrak{M}} : (\mathfrak{M}_3^\sharp, \preceq_3^{\mathfrak{M}}) \longrightarrow (\mathfrak{M}_2^\sharp, \preceq_2^{\mathfrak{M}})$ by putting $\gamma_3^{\mathfrak{M}} = \mathbf{UG}(\mathrm{Id}_{\mathbf{C}_1}, \kappa^3)$. We denote by $(\mathfrak{M}^\sharp, \preceq^{\mathfrak{M}})$ the domain $(\mathfrak{M}_3^\sharp, \preceq_3^{\mathfrak{M}})$ and by $\gamma^{\mathfrak{M}} : (\mathfrak{M}^\sharp, \preceq^{\mathfrak{M}}) \longrightarrow (\wp(\mathfrak{M}), \subseteq)$ the compound function $\gamma_0^{\mathfrak{M}} \circ \gamma_1^{\mathfrak{M}} \circ \gamma_2^{\mathfrak{M}} \circ \gamma_3^{\mathfrak{M}}$, which completes the construction of the domain of abstract memory configurations. It only remains to design widening operators on $\mathcal{D}^\sharp$ before describing the abstract semantics.

6 Widening Operators

The construction of widening operators is in general the critical part when designing a static analysis by abstract interpretation. Indeed, these operators strongly depend on the structure of the abstract semantic domain and there is thereby no general design methodology. A remarkable characteristic of cofibered domains is the existence of a systematic technique for constructing widening operators from elementary ones defined on the components of the domain. We will describe this technique and apply it to $(\mathcal{D}^\sharp, \preceq)$. We first need to extend the notion of widening to categories.

Definition 8 (Widening on a category) *A* widening operator ∇ *on a category* $\mathbf{C}$ *associates a pair* $A \overrightarrow{\nabla_1} B : A \longrightarrow A \nabla B$, $A \overrightarrow{\nabla_2} B : B \longrightarrow A \nabla B$ *of morphisms to any two objects* A *and* B *of* $\mathbf{C}$*, such that, for any sequence* $(A_n)_{n \geqslant 0}$ *of objects, the* ω*-chain* $(f_n^\nabla : A_n^\nabla \longrightarrow A_{n+1}^\nabla)_{n \geqslant 0}$ *inductively defined as follows:*

$$\begin{cases} A_0^\nabla & = A_0 \\ A_{n+1}^\nabla & = A_n^\nabla \nabla A_{n+1} \\ f_n^\nabla & = A_n^\nabla \overrightarrow{\nabla_1} A_{n+1} \end{cases}$$

is ultimately pseudo-stationary*, i.e. there exists a rank* $N \geqslant 0$ *such that for all* $n \geqslant N$, f_n^∇ *is an isomorphism. Moreover, we require* ∇ *to be* stable under isomorphism*, that is for any isomorphism* $\phi : A \longrightarrow A'$*, there exists an isomorphism* $\nabla\phi : A \nabla B \longrightarrow A' \nabla B$ *such that* $\nabla\phi \circ (A \overrightarrow{\nabla_1} B) = (A' \overrightarrow{\nabla_1} B) \circ \phi$*.*

Now, let $(D, \preceq)$ be a cofibered domain with display $\Delta : \mathbf{C} \longrightarrow \mathbf{Poset}$. We suppose that a widening operator ∇ is defined on $\mathbf{C}$ and that furthermore each fiber ΔA is provided with a widening ∇_A.

Theorem 9 (Widening on a cofibered domain) *The operator* $\nabla_{\mathbf{G}}$ *over* D *constructed as follows:*

(i) $(A, x) \nabla_{\mathbf{G}} (B, y) = \Big(A, x \nabla_A \Delta(A \overrightarrow{\nabla_1} B)^{-1} \circ \Delta(A \overrightarrow{\nabla_2} B)(y)\Big)$*, if* $A \overrightarrow{\nabla_1} B$ *is an isomorphism.*

(ii) $(A,x)\ \nabla_{\mathbf{G}}\ (B,y) = \Big(A \nabla B, \Delta(A\ \overrightarrow{\overline{\nabla}}_1\ B)(x)\ \nabla_{A\nabla B}\ \Delta(A\ \overrightarrow{\overline{\nabla}}_2\ B)(y)\Big)$, *otherwise.*

is a widening on D.

Intuitively, case (i) means that whenever the fiber is stable, i.e. $A\ \overrightarrow{\overline{\nabla}}_1\ B$ is an isomorphism, we transfer y into the fiber and we perform the widening with x. In the second case, we transfer x and y into the fiber over $A \nabla B$ and we make the widening therein.

Proof. Let $(A_n, x_n)_{n\geqslant 0}$ be a sequence of elements of D. Let $(f_n^\nabla : A_n^\nabla \longrightarrow A_{n+1}^\nabla)_{n\geqslant 0}$ be the ω-chain constructed from $(A_n)_{n\geqslant 0}$ following Definition 8. Let $(\overline{A}_n, \overline{x}_n)_{n\geqslant 0}$ be the sequence of elements of D inductively defined as $(\overline{A}_0, \overline{x}_0) = (A_0, x_0)$ and $(\overline{A}_{n+1}, \overline{x}_{n+1}) = (\overline{A}_n, \overline{x}_n)\ \nabla_{\mathbf{G}}\ (A_{n+1}, x_{n+1})$, for $n > 0$. We first show by induction on n that $A_n^\nabla \cong \overline{A}_n$ for every $n \geqslant 0$. It is obviously true for $n = 0$. We suppose that it is true for n. If $\overline{A}_{n+1} = \overline{A}_n$, this means that $\overline{A}_n\ \overrightarrow{\overline{\nabla}}_1\ A_{n+1}$ is an isomorphism. By induction hypothesis $A_n^\nabla \cong \overline{A}_n$, hence $\overline{A}_{n+1} = \overline{A}_n \cong \overline{A}_n \nabla A_{n+1} \cong A_n^\nabla \nabla A_{n+1} = A_{n+1}^\nabla$. If $\overline{A}_{n+1} = \overline{A}_n \nabla A_{n+1}$, then similarly $\overline{A}_{n+1} \cong A_n^\nabla \nabla A_{n+1} = A_{n+1}^\nabla$. By definition of ∇, there exists $N \geqslant 0$ such that f_n^∇ is an isomorphism for every $n \geqslant N$. We show by induction on n that $\overline{A}_n = \overline{A}_N$ for every $n \geqslant N$. This is obvious for $n = N$. We suppose that it is true for $n \geqslant N$. We have shown that there exists an isomorphism $\phi_n : A_n^\nabla \longrightarrow \overline{A}_n$. Following Definition 8, we have $(\overline{A}_n\ \overrightarrow{\overline{\nabla}}_1\ A_{n+1}) \circ \phi_n = \nabla\phi_n \circ f_n^\nabla$. But since f_n^∇ is an isomorphism, $\overline{A}_n \overrightarrow{\overline{\nabla}}_1 A_{n+1}$ is also an isomorphism. Then, by definition of $\nabla_{\mathbf{G}}$, $\overline{A}_{n+1} = \overline{A}_n$, and by induction hypothesis $\overline{A}_{n+1} = \overline{A}_N$. Since for any $n \geqslant N$, $\overline{A}_n = \overline{A}_N$, it follows from the definition of $\nabla_{\mathbf{G}}$ that there exists a sequence $(\overline{\overline{x}}_n)_{n>N}$ of elements of $\Delta\overline{A}_N$, such that $\overline{x}_{n+1} = \overline{x}_n\ \nabla_{\overline{A}_N}\ \overline{\overline{x}}_{n+1}$, for every $n \geqslant N$. But $\nabla_{\overline{A}_N}$ is a widening operator on $\Delta\overline{A}_N$, hence there exists $M \geqslant N$ such that $\overline{x}_n = \overline{x}_M$, for every $n \geqslant M$. Therefore, for every $n \geqslant M$, $(\overline{A}_n, \overline{x}_n) = (\overline{A}_M, \overline{x}_M)$, which concludes the proof. □

Since $\mathcal{D}^\sharp$ is the product $\prod_{\mathfrak{p}\in\mathfrak{P}} \mathfrak{M}^\sharp$, it is sufficient to construct a widening operator on the cofibered domain $\mathfrak{M}^\sharp$ and to apply it pointwise to elements of $\mathcal{D}^\sharp$. All abstract numerical domains $\mathcal{N}$ come with a widening operator ∇_V defined on every lattice $\mathcal{N}_V$. For example, in Karr's domain of linear equalities all lattices $\mathcal{N}_V$ have finite height, hence we can take the join $\sqcup_V$ as a widening. Therefore, the fiber of $\mathfrak{M}^\sharp$ over a tuple of automata $\mathbf{A}$ being given by $\prod_{\varkappa\in\hat{\mathbf{A}}} \mathcal{N}_{\mathrm{C}(\varkappa)}$, the pointwise application of operators $\nabla_{\mathrm{C}(\varkappa)}$ provides us with a widening operator on that fiber. In order to apply Theorem 9, we must also define a widening operation ∇ on the base category $\mathbf{C}_1$ of $\mathfrak{M}^\sharp$. Since $\mathbf{C}_1$ is the product category $\prod_{x\in\mathfrak{V}} \mathbf{Aut}$, we only need to construct a widening operator on $\mathbf{Aut}$ and to apply it pointwise to objects of $\mathbf{C}_1$.

Fortunately, we do not have to start from scratch, since several widening techniques for automata [3] have already been devised [VCL94,GdW94,CC95b] and can be applied here. However, these methods are all rather involved and tedious to describe. Hence, for explanatory purposes we present an extremely simple widening operation, which is nevertheless accurate enough to illustrate our alias analysis on the program of Figure 1. The idea is to limit the size of an automaton by requiring each data selector $\sigma \in \Sigma$ to be carried by at most one transition of the automaton [4].

More formally, let $\mathcal{A}_1 = (Q_1, i_1, T_2)$ and $\mathcal{A}_2 = (Q_2, i_2, T_2)$ be two deterministic automata. We suppose that Q_1 and Q_2 are disjoint, which is always possible up to bijective state renaming. Let $Q = Q_1 \cup Q_2$, $T = T_1 \cup T_2$ and $\sim$ be an equivalence relation on Q. We say that $\sim$ is *admissible* if the following conditions are satisfied:

- $i_1 \sim i_2$
- $\forall (q_1, \sigma, q_2), (q_1', \sigma', q_2') \in T : q_1 \sim q_1' \wedge \sigma = \sigma' \Longrightarrow q_2 \sim q_2'$

If $\sim$ is admissible, we define the $\sim$-*join* of $\mathcal{A}_1$ and $\mathcal{A}_2$ as:

$$\mathcal{A}_1 \,\square_\sim\, \mathcal{A}_2 = (Q/_\sim, [i_1]_\sim, \{([q]_\sim, \sigma, [q']_\sim) \mid (q, \sigma, q') \in T\})$$

This definition is consistent, since the automaton we obtain is clearly deterministic. Moreover, $\mathcal{A}_1$ and $\mathcal{A}_2$ can be canonically embedded in $\mathcal{A}_1 \,\square_\sim\, \mathcal{A}_2$ via the morphisms which map any state q to its equivalence class $[q]_\sim$. If $\sim$ is the least admissible relation on Q, we will denote by $\mathcal{A}_1 \,\square\, \mathcal{A}_2$ the $\sim$-join of $\mathcal{A}_1$ and $\mathcal{A}_2$, that we will simply call the *join*.

Now, we denote by $\sim_\nabla$ the least admissible equivalence relation on Q satisfying the following condition:

$$\forall (q_1, \sigma, q_2), (q_1', \sigma', q_2') \in T : \sigma = \sigma' \Longrightarrow (q_1 \sim q_1' \wedge q_2 \sim q_2')$$

We define $\mathcal{A}_1 \,\nabla\, \mathcal{A}_2$ as the $\sim_\nabla$-join of $\mathcal{A}_1$ and $\mathcal{A}_2$. We readily check that every data selector $\sigma \in \Sigma$ is carried by at most one transition of $\mathcal{A}_1 \nabla \mathcal{A}_2$. Since there are finitely many nonisomorphic automata satisfying this property, we obtain a widening operator. The associated morphisms $\mathcal{A}_1 \,\overrightarrow{\nabla_1}\, \mathcal{A}_2 : \mathcal{A}_1 \longrightarrow \mathcal{A}_1 \,\nabla\, \mathcal{A}_2$ and $\mathcal{A}_1 \,\overrightarrow{\nabla_2}\, \mathcal{A}_2 : \mathcal{A}_2 \longrightarrow \mathcal{A}_1 \nabla \mathcal{A}_2$ are just the canonical embeddings. Moreover, this widening operator is clearly stable under isomorphism.

[3] These methods have originally been designed for the larger class of *tree automata* [GS84].

[4] The idea of this widening operation originated from a discussion with M. Felleisen.

7 Abstract Semantics of the Language

In order to define the abstract semantics of our small language we first need to introduce some basic operations. Let $(\mathbf{A}_1, \mathbf{R}_1^\sharp)$ and $(\mathbf{A}_2, \mathbf{R}_2^\sharp)$ be two abstract memory configurations of $\mathfrak{M}^\sharp$. Their *join* $(\mathbf{A}_1, \mathbf{R}_1^\sharp) \oplus (\mathbf{A}_2, \mathbf{R}_2^\sharp)$ is the abstract memory configuration $(\mathbf{A}, \mathbf{R}^\sharp)$ defined as follows. For all $x \in \mathfrak{V}$, $\mathbf{A}(x) = \mathbf{A}_1(x) \sqcap \mathbf{A}_2(x)$. Let $\mathbf{f} : \mathbf{A}_1 \longrightarrow \mathbf{A}$ and $\mathbf{g} : \mathbf{A}_2 \longrightarrow \mathbf{A}$ be the arrows of $\mathbf{C}_1$ induced by the canonical embedding morphisms. For every $\varkappa \in \hat{\mathbf{A}}$, we put

$$\mathbf{R}^\sharp(\varkappa) = (\Delta_3 \mathbf{f}(\mathbf{R}_1^\sharp))(\varkappa) \sqcup_{\mathrm{C}(\varkappa)} (\Delta_3 \mathbf{g}(\mathbf{R}_2^\sharp))(\varkappa)$$

Then, by construction we have $(\mathbf{A}_1, \mathbf{R}_1^\sharp) \preceq^{\mathfrak{M}} (\mathbf{A}, \mathbf{R}^\sharp)$ and $(\mathbf{A}_2, \mathbf{R}_2^\sharp) \preceq^{\mathfrak{M}} (\mathbf{A}, \mathbf{R}^\sharp)$. Note that the operation $\oplus$ is associative and commutative up to isomorphisms of automata. Therefore, it will make sense to write the join of a nonempty family of abstract memory configurations. Now, let $q_\perp$ be a state in $\mathcal{Q}$. We denote by $\mathcal{A}_\perp$ the automaton $(\{q_\perp\}, q_\perp, \emptyset)$ recognizing the singleton $\{\varepsilon\}$. Let $\mathfrak{m}_\perp^\sharp$ be the abstract memory configuration $(\mathbf{A}_\perp, \mathbf{R}_\perp^\sharp)$ where:

- $\forall x \in \mathfrak{V} : \mathbf{A}_\perp(x) = \mathcal{A}_\perp$
- $\forall \varkappa \in \hat{\mathbf{A}}_\perp : \mathbf{R}_\perp^\sharp(\varkappa) = \perp_{\mathrm{C}(\varkappa)}$

By convention, the join of an empty family of abstract memory configurations is equal to $\mathfrak{m}_\perp^\sharp$.

We also need two additional operations on the abstract numerical domain: the *projection* $\overleftarrow{\mathcal{N}}$ and the *extension* $\overrightarrow{\mathcal{N}}$. Let U and V be finite sets of variables such that $U \subseteq V$. For every $\nu \in \prod_{v \in V} \mathbb{N}$, we denote by $\nu|_U$ the *restriction* of ν to $\prod_{u \in U} \mathbb{N}$, that is, for every $u \in U$, we have $\nu|_U(u) = \nu(u)$. Every abstract numerical domain comes with two functions $\overleftarrow{\mathcal{N}}_{U,V} : \mathcal{N}_V \longrightarrow \mathcal{N}_U$ and $\overrightarrow{\mathcal{N}}_{U,V} : \mathcal{N}_U \longrightarrow \mathcal{N}_V$ satisfying the following conditions:

- $\forall N \in \mathcal{N}_V : \{\nu|_U \mid \nu \in \gamma_V(N)\} \subseteq \gamma_U(\overleftarrow{\mathcal{N}}_{U,V}(\nu))$.
- $\forall N' \in \mathcal{N}_U : \{\nu \in \mathcal{N}_V \mid \nu|_U \in \gamma_U(N')\} \subseteq \gamma_V(\overrightarrow{\mathcal{N}}_{U,V}(N'))$.

Example 10 *We consider Karr's domain of linear equalities. If $N \in \mathcal{N}_V$ is generated by the system of vectors $\{e_1, \ldots, e_n\}$, then $\overleftarrow{\mathcal{N}}_{U,V}(N)$ is generated by the system $\{e_1|_U, \ldots, e_n|_U\}$. If $N' \in \mathcal{N}_U$, then N' is the solution of a system of linear equalities S on U. $\overrightarrow{\mathcal{N}}_{U,V}(N')$ is the linear variety which is solution of S on V.*

The function $\mathrm{F}^\sharp$ will be defined by assigning an abstract semantics $[\![\mathfrak{i}]\!]^\sharp : \mathfrak{M}^\sharp \longrightarrow \mathfrak{M}^\sharp$ to each instruction $\mathfrak{i}$ of the language, such that the fol-

lowing soundness condition holds:

$$\forall \mathfrak{m}^\sharp \in \mathfrak{M}^\sharp : \forall \mathfrak{m} \in \gamma^{\mathfrak{M}}(\mathfrak{m}^\sharp) : [\![\mathfrak{i}]\!]\mathfrak{m} \in \gamma^{\mathfrak{M}}([\![i]\!]^\sharp \mathfrak{m}^\sharp)$$

Then, we can define the abstract semantic function $\mathrm{F}^\sharp : \mathcal{D}^\sharp \longrightarrow \mathcal{D}^\sharp$ as follows:

$$\forall C^\sharp \in \mathcal{D}^\sharp : \forall \mathfrak{p} \in \mathfrak{P} : \mathrm{F}^\sharp(C^\sharp)(\mathfrak{p}) = \bigoplus_{\mathfrak{p}' \xrightarrow{\mathfrak{i}} \mathfrak{p}} [\![\mathfrak{i}]\!]^\sharp C^\sharp(\mathfrak{p}')$$

Assuming the soundness condition holds for every instruction of the language, we can prove without any difficulty that the definition of $\mathrm{F}^\sharp$ is consistent:

Theorem 11 $\forall C^\sharp \in \mathcal{D}^\sharp : \mathrm{F} \circ \gamma(C^\sharp) \subseteq \gamma \circ \mathrm{F}^\sharp(C^\sharp)$.

If $\mathbf{A}$ is a tuple of automata of $\mathbf{C}_1$, and $\varkappa = \langle (x, q), (y, q') \rangle$ is an element of $\hat{\mathbf{A}}$, we denote by $\mathfrak{V}(\varkappa)$ the set $\{x, y\}$. Now, let $x := \mathbf{new}\, \mathfrak{r}$ be a record allocation instruction, with $\mathfrak{F}(\mathfrak{r}) = \{\mathfrak{f}_1, \ldots, \mathfrak{f}_n\}$. The abstract semantics of this instruction mimics the concrete one, i.e. we remove all access paths starting from x, as well as all related alias pairs, and we add the access paths corresponding to the fields of record $\mathfrak{r}$, the newly created access paths being left unaliased. More precisely, let $q, q_1, \ldots, q_n$ be distinct states of $\mathcal{Q}$. If $\mathfrak{m}^\sharp = (\mathbf{A}, \mathbf{R}^\sharp)$ is an abstract memory configuration, we put $[\![x := \mathbf{new}\, \mathfrak{r}]\!]^\sharp \mathfrak{m}^\sharp = (\mathbf{A}_\bullet, \mathbf{R}^\sharp_\bullet)$, where:

- $\forall z \in \mathfrak{V} : \mathbf{A}_\bullet(z) = \begin{cases} (\{q, q_1, \ldots, q_n\}, q, \{(q, \mathfrak{r}\sharp\mathfrak{f}_i, q_i) \mid 1 \leqslant i \leqslant n\}) & \text{if } z = x \\ \mathbf{A}(z) & \text{otherwise} \end{cases}$
- $\forall \varkappa \in \hat{\mathbf{A}}_\bullet : \mathbf{R}^\sharp_\bullet(\varkappa) = \begin{cases} \bot_{\mathrm{C}(\varkappa)} & \text{if } x \in \mathfrak{V}(\varkappa) \\ \mathbf{R}^\sharp(\varkappa) & \text{otherwise} \end{cases}$

It is straightforward to check the soundness of this definition:

Theorem 12 *For all* $\mathfrak{m}^\sharp \in \mathfrak{M}^\sharp$ *and* $\mathfrak{m} \in \gamma^{\mathfrak{M}}(\mathfrak{m}^\sharp)$, *we have*

$$[\![x := \mathbf{new}\, \mathfrak{r}]\!]\mathfrak{m} \in \gamma^{\mathfrak{M}}([\![x := \mathbf{new}\, \mathfrak{r}]\!]^\sharp \mathfrak{m}^\sharp)$$

All assignment instructions are based upon the **set** operation, which is itself defined by using the closure operation $\varrho_{\mathfrak{M}}$. The latter can be expressed as the computation of a least fixpoint over the domain $\mathfrak{M}^\sharp_0$ endowed with its canonical structure $(\mathfrak{M}^\sharp_0, \preceq^{\mathfrak{M}}_0, \sqcup_{\mathfrak{M}^\sharp_0}, \bot_{\mathfrak{M}^\sharp_0}, \sqcap_{\mathfrak{M}^\sharp_0}, \top_{\mathfrak{M}^\sharp_0})$ of complete lattice. Indeed, let $\mathrm{F_s}$,$\mathrm{F_t}$, $\mathrm{F_{rr1}}$ and $\mathrm{F_{rr2}}$ be the endofunctions of $\mathfrak{M}^\sharp_0$ such that, for every (Π, ρ) in $\mathfrak{M}^\sharp_0$, we have:

- $\mathrm{F_s}(\Pi, \rho) = (\Pi, \rho \cup \{(\pi_2, \pi_1) \mid (\pi_1, \pi_2) \in \rho\})$

– $\mathrm{F_t}(\Pi, \rho) = (\Pi, \rho \cup \{(\pi_1, \pi_3) \mid \exists \pi_2 \in \Pi : (\pi_1, \pi_2) \in \rho \wedge (\pi_2, \pi_3) \in \rho\})$
– $\mathrm{F_{rr1}}(\Pi, \rho) = (\Pi, \rho \cup \{(\pi_1.\sigma, \pi_2.\sigma) \mid (\pi_1, \pi_2) \in \rho \wedge \pi_1.\sigma, \pi_2.\sigma \in \Pi\})$
– $\mathrm{F_{rr2}}(\Pi, \rho) = (\Pi \cup \{\pi_1.\sigma \mid \exists \pi_2 \in \Pi : (\pi_1, \pi_2) \in \rho \wedge \pi_2.\sigma \in \Pi\}, \rho)$

Let F_ϱ be the $\sqcup_{\mathfrak{M}_0^\sharp}$-complete and extensive endomorphism of $\mathfrak{M}_0^\sharp$ such that, for any (Π, ρ) in $\mathfrak{M}_0^\sharp$, we have:

$$\mathrm{F}_\varrho(\Pi, \rho) = \mathrm{F_s}(\Pi, \rho) \sqcup_{\mathfrak{M}_0^\sharp} \mathrm{F_t}(\Pi, \rho) \sqcup_{\mathfrak{M}_0^\sharp} \mathrm{F_{rr1}}(\Pi, \rho) \sqcup_{\mathfrak{M}_0^\sharp} \mathrm{F_{rr2}}(\Pi, \rho)$$

Now, let $(\Pi', \rho') = \mathrm{lfp}_{(\Pi,\rho)} \mathrm{F}_\varrho$ be the least fixpoint of F_ϱ greater than or equal to (Π, ρ). Then, using standard results [CC95a], we can easily prove that $\varrho_{\mathfrak{M}}(\Pi, \rho)$ is equal to $(\Pi', \rho' \cup \mathrm{D}(\Pi'))$. Each of the functions $\mathrm{F_s}$, $\mathrm{F_t}$, $\mathrm{F_{rr1}}$ and $\mathrm{F_{rr2}}$ actually encodes a certain part of the closure operation (symmetry, transitivity and both aspects of right-regularity), reflexivity being deduced from the set of access paths. We will construct an abstract counterpart of F_ϱ over $\mathfrak{M}^\sharp$ and compute an approximation of $\varrho_{\mathfrak{M}}(\Pi, \rho)$ by using the techniques of Sect. 3.

We denote by $\gamma_{3,0}^{\mathfrak{M}} : \mathfrak{M}^\sharp \longrightarrow \mathfrak{M}_0^\sharp$ the compound function $\gamma_1^{\mathfrak{M}} \circ \gamma_2^{\mathfrak{M}} \circ \gamma_3^{\mathfrak{M}}$ (recall that we have denoted by $\mathfrak{M}^\sharp$ the domain $\mathfrak{M}_3^\sharp$ of abstract memory configurations). We introduce the endofunctions $\mathrm{F}_\mathrm{s}^\sharp$, $\mathrm{F}_\mathrm{t}^\sharp$, $\mathrm{F}_\mathrm{rr1}^\sharp$ and $\mathrm{F}_\mathrm{rr2}^\sharp$ over $\mathfrak{M}^\sharp$ which are sound approximations of the corresponding functions over $\mathfrak{M}_0^\sharp$. Let $(\mathbf{A}, \mathbf{R}^\sharp)$ be an element of $\mathfrak{M}^\sharp$. We mimic the exact functions, replacing every occurrence of a path by its numerical abstraction and adapting the definition to the partitioning given by $\hat{\mathbf{A}}$.

Symmetry. $\mathrm{F}_\mathrm{s}^\sharp(\mathbf{A}, \mathbf{R}^\sharp)$ is the abstract memory configuration $(\mathbf{A}, \mathbf{R}_\mathrm{s}^\sharp)$ defined as follows. Let $\varkappa = \langle (x, q), (y, q') \rangle$ be an element of $\hat{\mathbf{A}}$. We put $\varkappa' = \langle (y, q'), (x, q) \rangle$. Let $s : \prod_{k \in \mathrm{C}(\varkappa')} \mathbb{N} \longrightarrow \prod_{k \in \mathrm{C}(\varkappa)} \mathbb{N}$ be the linear map which sends any ν to the tuple of integers ν' such that:

– $\forall \mathbf{l}.c \in \mathrm{C}(\varkappa) : \nu'(\mathbf{l}.C) = \nu(\mathbf{r}.c)$
– $\forall \mathbf{r}.c \in \mathrm{C}(\varkappa) : \nu'(\mathbf{r}.C) = \nu(\mathbf{l}.c)$

This map simply allows us to permute the values of the left and right counters. Then, we have

$$\mathbf{R}_\mathrm{s}^\sharp(\varkappa) = \mathbf{R}^\sharp(\varkappa) \sqcup_{\mathrm{C}(\varkappa)} \mathcal{N}s(\mathbf{R}^\sharp(\varkappa'))$$

Transitivity. $\mathrm{F}_\mathrm{t}^\sharp(\mathbf{A}, \mathbf{R}^\sharp)$ is the abstract memory configuration $(\mathbf{A}, \mathbf{R}_\mathrm{t}^\sharp)$ defined as follows. Let $\varkappa = \langle (x, q), (z, q'') \rangle$ be an element of $\hat{\mathbf{A}}$. If $y \in \mathfrak{V}$, we denote by Q_y the set of states of $\mathbf{A}(y)$. Now, let $q' \in Q_y$, $\varkappa_1 = \langle (x, q), (y, q') \rangle$ and $\varkappa_2 = \langle (y, q'), (z, q'') \rangle$. For each $c \in \mathrm{C}(\mathbf{A}(y))$, we take a fresh variable v_c. Let $V_1 = \{\mathbf{l}.c \mid c \in \mathrm{C}(\mathbf{A}(x))\} \cup \{v_c \mid c \in \mathrm{C}(\mathbf{A}(y))\}$, $V_2 = \{\mathbf{r}.c \mid c \in \mathrm{C}(\mathbf{A}(z))\} \cup \{v_c \mid$

$c \in \mathrm{C}(\mathbf{A}(y))\}$ and $V = V_1 \cup V_2$. Let $t_1 : \prod_{k \in \mathrm{C}(\varkappa_1)} \mathbb{N} \longrightarrow \prod_{v \in V_1} \mathbb{N}$ be the linear map which sends any ν to the tuple of integers ν' such that:

- $\forall \mathbf{l}.c \in V_1 : \nu'(\mathbf{l}.c) = \nu(\mathbf{l}.c)$
- $\forall v_c \in \mathrm{V}_1 : \nu'(v_c) = \nu(\mathbf{r}.c)$

Similarly, let $t_2 : \prod_{k \in \mathrm{C}(\varkappa_2)} \mathbb{N} \longrightarrow \prod_{v \in V_2} \mathbb{N}$ be the linear map which sends any ν to the tuple of integers ν' such that:

- $\forall \mathbf{r}.c \in V_2 : \nu'(\mathbf{r}.c) = \nu(\mathbf{r}.c)$
- $\forall v_c \in \mathrm{V} : \nu'(v_c) = \nu(\mathbf{l}.c)$

We denote by $\mathrm{t}_{y,q'}(\varkappa)$ the element of $\mathcal{N}_{\mathrm{C}(\varkappa)}$ defined as follows:

$$\mathrm{t}_{y,q'}(\varkappa) = \overleftarrow{\mathcal{N}}_{\mathrm{C}(\varkappa),V} \left(\overrightarrow{\mathcal{N}}_{V_1,V} \left(\mathcal{N} t_1(\mathbf{R}^\sharp(\varkappa_1)) \right) \sqcap_V \overrightarrow{\mathcal{N}}_{V_2,V} \left(\mathcal{N} t_2(\mathbf{R}^\sharp(\varkappa_2)) \right) \right)$$

This apparently complicated formula simply computes an approximation of the set of elements ν of $\mathcal{N}_{\mathrm{C}(\varkappa)}$ for which there exist ν_1 in $\mathcal{N}_{\mathrm{C}(\varkappa_1)}$ and ν_2 in $\mathcal{N}_{\mathrm{C}(\varkappa_2)}$ such that the following conditions hold:

- $\forall c \in \mathrm{C}(\mathbf{A}(y)) : \nu_1(\mathbf{r}.c) = \nu_2(\mathbf{l}.c)$
- $\forall c \in \mathrm{C}(\mathbf{A}(x)) : \nu(\mathbf{l}.c) = \nu_1(\mathbf{l}.c)$
- $\forall c \in \mathrm{C}(\mathbf{A}(z)) : \nu(\mathbf{r}.c) = \nu_2(\mathbf{r}.c)$

Then, we have

$$\mathbf{R}^\sharp_{\mathrm{t}}(\varkappa) = \mathbf{R}^\sharp(\varkappa) \sqcup_{\mathrm{C}(\varkappa)} \bigsqcup\nolimits_{\mathrm{C}(\varkappa)} \{ \mathrm{t}_{y,q'}(\varkappa) \mid y \in \mathfrak{V} \wedge q' \in Q_y \}$$

Right-regularity 1. $\mathrm{F}^\sharp_{\mathrm{rr1}}(\mathbf{A}, \mathbf{R}^\sharp)$ is the abstract memory configuration $(\mathbf{A}, \mathbf{R}^\sharp_{\mathrm{rr1}})$ defined as follows. Let $\varkappa_1 = \langle (x, q_1), (y, q'_1) \rangle$ and $\varkappa_2 = \langle (x, q_2), (y, q'_2) \rangle$ be two elements of $\hat{\mathbf{A}}$. Note that $\mathrm{C}(\varkappa_1) = \mathrm{C}(\varkappa_2)$. For any $\sigma \in \Sigma$, we say that $\varkappa_2 = \varkappa_1.\sigma$ whenever there exists a transition $q_1 \xrightarrow{\sigma} q_2$ in $\mathbf{A}(x)$ and $q'_1 \xrightarrow{\sigma} q'_2$ in $\mathbf{A}(y)$. If $\varkappa_2 = \varkappa_1.\sigma$, we denote by $i_{\varkappa_1,\sigma} : \prod_{k \in \mathrm{C}(\varkappa_2)} \mathbb{N} \longrightarrow \prod_{k \in \mathrm{C}(\varkappa_2)} \mathbb{N}$ the linear map which sends any ν to the tuple of integers ν' such that:

- $\nu'(\mathbf{l}.q_1.\sigma) = \nu(\mathbf{l}.q_1.\sigma) + 1$
- $\nu'(\mathbf{r}.q'_1.\sigma) = \nu(\mathbf{r}.q'_1.\sigma) + 1$
- $\nu'(k) = \nu(k)$ for any other counter k

This map simply encodes the operation which consists of simultaneously incrementing counters $\mathbf{l}.q_1.\sigma$ and $\mathbf{r}.q'_1.\sigma$, leaving the value of all other counters unchanged. Then, for all $\varkappa \in \hat{\mathbf{A}}$, we have

$$\mathbf{R}^\sharp_{\mathrm{rr1}}(\varkappa) = \mathbf{R}^\sharp(\varkappa) \sqcup_{\mathrm{C}(\varkappa)} \bigsqcup\nolimits_{\mathrm{C}(\varkappa)} \{ i_{\varkappa',\sigma}(\mathbf{R}^\sharp(\varkappa')) \mid \sigma \in \Sigma \wedge \varkappa = \varkappa'.\sigma \}$$

Right-regularity 2. $\mathrm{F}^\sharp_{\mathrm{rr2}}(\mathbf{A}, \mathbf{R}^\sharp)$ is the abstract memory configuration $(\mathbf{A}_{\mathrm{rr2}}, \mathbf{R}^\sharp_{\mathrm{rr2}})$ defined as follows. Let $x \in \mathfrak{V}$. We put $\mathbf{A}(x) = (Q_x, i_x, T_x)$. Now, let $q \in Q_x$ and $\sigma \in \Sigma$. If there exists a transition $q \xrightarrow{\sigma} q'$ in $\mathbf{A}(x)$, we denote by q_σ the state q'. Otherwise, if there exists $\varkappa = \langle (x, q), (y, q') \rangle \in \hat{\mathbf{A}}$ such that $\mathbf{R}^\sharp(\varkappa) \neq \bot_{\mathrm{C}(\varkappa)}$ and there exists a transition $q' \xrightarrow{\sigma} q''$ in $\mathbf{A}(y)$, we denote by q_σ a fresh state of $\mathcal{Q}$. Then, we have $\mathbf{A}_{\mathrm{rr2}}(x) = (Q'_x, i_x, T'_x)$, where

- $Q'_x = Q_x \cup \{q_\sigma \mid q \in Q_x \wedge \sigma \in \Sigma\}$
- $T'_x = T_x \cup \{(q, \sigma, q_\sigma) \mid q \in Q_x \wedge \sigma \in \Sigma\}$

Thus, we have extended each automaton of $\mathbf{A}$ in order to take the transitions induced by the aliasing relation into account. Note that each automaton $\mathbf{A}(x)$ embeds in $\mathbf{A}_{\mathrm{rr2}}(x)$. Let $\mathbf{i} : \mathbf{A} \longrightarrow \mathbf{A}_{\mathrm{rr2}}$ be the tuple of all these embedding morphisms. Then, we have

$$\mathbf{R}^\sharp_{\mathrm{rr2}} = \Delta_3 \mathbf{i}(\mathbf{R}^\sharp)$$

Let $\mathrm{F}^\sharp_\varrho$ be the endofunction of $\mathfrak{M}^\sharp$ such that, for any $\mathfrak{m}^\sharp \in \mathfrak{M}^\sharp$, we have

$$\mathrm{F}^\sharp_\varrho(\mathfrak{m}^\sharp) = \mathrm{F}^\sharp_{\mathrm{s}}(\mathfrak{m}^\sharp) \oplus \mathrm{F}^\sharp_{\mathrm{t}}(\mathfrak{m}^\sharp) \oplus \mathrm{F}^\sharp_{\mathrm{rr1}}(\mathfrak{m}^\sharp) \oplus \mathrm{F}^\sharp_{\mathrm{rr2}}(\mathfrak{m}^\sharp)$$

Theorem 13 *For any $\mathfrak{m}^\sharp \in \mathfrak{M}^\sharp$, we have*

$$\mathrm{F}_\varrho \circ \gamma^{\mathfrak{M}}_{3,0}(\mathfrak{m}^\sharp) \preceq^{\mathfrak{M}}_0 \gamma^{\mathfrak{M}}_{3,0} \circ \mathrm{F}^\sharp_\varrho(\mathfrak{m}^\sharp)$$

Despite the apparent complexity of the abstract closure functions, this soundness result is trivial. It is an immediate consequence of the construction of these functions. For any $\mathfrak{m}^\sharp$ in $\mathfrak{M}^\sharp$, we denote by $\varrho^\sharp_{\mathfrak{M}}(\mathfrak{m}^\sharp)$ the limit of the abstract iteration sequence defined in Sect. 3 using $\mathrm{F}^\sharp_\varrho$ as the abstract semantic function and $\mathfrak{m}^\sharp$ as the abstract basis.

Now, let $(\mathbf{A}, \mathbf{R}^\sharp)$ be an abstract memory configuration. The abstract semantics of instructions $x := y$ and $x := y.\mathfrak{r}\sharp\mathfrak{f}$ go along the same lines. Let $(\mathbf{A}_\bullet, \mathbf{R}^\sharp_\bullet)$ be the element of $\mathfrak{M}^\sharp$ defined as follows:

- $\forall z \in \mathfrak{V} : \mathbf{A}_\bullet(z) = \begin{cases} \mathcal{A}_\bot & \text{if } z = x \\ \mathbf{A}(z) & \text{otherwise} \end{cases}$
- $\forall \varkappa \in \hat{\mathbf{A}}_\bullet : \mathbf{R}^\sharp_\bullet(\varkappa) = \begin{cases} \bot_{\mathrm{C}(\varkappa)} & \text{if } x \in \mathfrak{V}(\varkappa) \\ \mathbf{R}^\sharp(\varkappa) & \text{otherwise} \end{cases}$

We denote by i_y the initial state of $\mathbf{A}_\bullet(y)$.

Variable aliasing. Let $\varkappa_{x,y}$ be the element $\langle (x, q_\bot), (y, i_y) \rangle$ of $\hat{\mathbf{A}}_\bullet$. We denote by $f_0 : \prod_{k \in \mathrm{C}(\varkappa_{x,y})} \mathbb{N} \longrightarrow \prod_{k \in \mathrm{C}(\varkappa_{x,y})} \mathbb{N}$ the linear map which sends any ν to the

tuple of integers ν' such that:

$$\forall k \in \mathrm{C}(\varkappa_{x,y}) : \nu'(k) = 0$$

The effect of this function is to set the value of all counters in $\mathrm{C}(\varkappa_{x,y})$ to 0. Let $\overline{\mathbf{R}}^\sharp_\bullet$ be the abstract aliasing relation over $\mathbf{A}_\bullet$ such that:

$$\forall \varkappa \in \hat{\mathbf{A}}_\bullet : \overline{\mathbf{R}}^\sharp_\bullet(\varkappa) = \begin{cases} \mathcal{N}f_0(\top_{\mathrm{C}(\varkappa_{x,y})}) & \text{if } \varkappa = \varkappa_{x,y} \\ \mathbf{R}^\sharp_\bullet(\varkappa) & \text{otherwise} \end{cases}$$

Then, we put

$$[\![x := y]\!]^\sharp(\mathbf{A}, \mathbf{R}^\sharp) = \varrho^\sharp_{\mathfrak{M}}(\mathbf{A}_\bullet, \overline{\mathbf{R}}^\sharp_\bullet)$$

Pointer assignment. If there is no transition labelled by $\mathfrak{r}\sharp\mathfrak{f}$ originating from i_y, this corresponds to accessing to uninitialized memory, which is a runtime error. Therefore, we put

$$[\![x := y.\mathfrak{r}\sharp\mathfrak{f}]\!]^\sharp(\mathbf{A}, \mathbf{R}^\sharp) = \mathfrak{m}^\sharp_\perp$$

Otherwise, let q be the state of $\mathbf{A}(y)$ such that $i_y \xrightarrow{\mathfrak{r}\sharp\mathfrak{f}} q$. Let $\varkappa'_{x,y}$ be the element $\langle (x, q_\perp), (y, q) \rangle$ of $\hat{\mathbf{A}}_\bullet$. We denote by $f_1 : \prod_{k \in \mathrm{C}(\varkappa'_{x,y})} \mathbb{N} \longrightarrow \prod_{k \in \mathrm{C}(\varkappa'_{x,y})} \mathbb{N}$ the linear map which sends any ν to the tuple of integers ν' such that:

$$\forall k \in \mathrm{C}(\varkappa'_{x,y}) : \nu'(k) = \begin{cases} 1 & \text{if } k = \mathbf{r}.i_y.\mathfrak{r}\sharp\mathfrak{f} \\ 0 & \text{otherwise} \end{cases}$$

The effect of this function is to set the value of all counters in $\mathrm{C}(\varkappa'_{x,y})$ to 0, except $\mathbf{r}.i_y.\mathfrak{r}\sharp\mathfrak{f}$ which is set to 1. Let $\overline{\overline{\mathbf{R}}}^\sharp_\bullet$ be the abstract aliasing relation over $\mathbf{A}_\bullet$ such that:

$$\forall \varkappa \in \hat{\mathbf{A}}_\bullet : \overline{\overline{\mathbf{R}}}^\sharp_\bullet(\varkappa) = \begin{cases} \mathcal{N}f_1(\top_{\mathrm{C}(\varkappa'_{x,y})}) & \text{if } \varkappa = \varkappa'_{x,y} \\ \mathbf{R}^\sharp_\bullet(\varkappa) & \text{otherwise} \end{cases}$$

Then, we put

$$[\![x := y.\mathfrak{r}\sharp\mathfrak{f}]\!]^\sharp(\mathbf{A}, \mathbf{R}^\sharp) = \varrho^\sharp_{\mathfrak{M}}(\mathbf{A}_\bullet, \overline{\overline{\mathbf{R}}}^\sharp_\bullet)$$

Destructive assignment. We finally consider the case of a destructive assignment instruction $x.\mathfrak{r}\sharp\mathfrak{f} := y$. We denote by i_x the initial state of $\mathbf{A}(x)$. If there is no transition labelled by $\mathfrak{r}\sharp\mathfrak{f}$ originating from i_x, this corresponds to accessing to uninitialized memory. Therefore, we put

$$[\![x.\mathfrak{r}\sharp\mathfrak{f} := y]\!]^\sharp(\mathbf{A}, \mathbf{R}^\sharp) = \mathfrak{m}^\sharp_\bot$$

Otherwise, let q be the state of $\mathbf{A}(x)$ such that $i_x \xrightarrow{\mathfrak{r}\sharp\mathfrak{f}} q$. Let $\varkappa''_{x,y}$ be the element $\langle (x,q),(y,i_y)\rangle$ of $\hat{\mathbf{A}}$. We denote by $g_1 : \prod_{k\in \mathrm{C}(\varkappa''_{x,y})} \mathbb{N} \longrightarrow \prod_{k\in \mathrm{C}(\varkappa''_{x,y})} \mathbb{N}$ the linear map which sends any ν to the tuple of integers ν' such that:

$$\forall k \in \mathrm{C}(\varkappa''_{x,y}) : \nu'(k) = \begin{cases} 1 & \text{if } k = \mathbf{l}.i_x.\mathfrak{r}\sharp\mathfrak{f} \\ 0 & \text{otherwise} \end{cases}$$

The effect of this function is to set the value of all counters to 0, except $\mathbf{l}.i_x.\mathfrak{r}\sharp\mathfrak{f}$ which is set to 1. Let $\mathbf{R}^\sharp_\dagger$ be the abstract aliasing relation over $\mathbf{A}$ such that:

$$\forall \varkappa \in \hat{\mathbf{A}} : \mathbf{R}^\sharp_\dagger(\varkappa) = \begin{cases} \mathbf{R}^\sharp(\varkappa''_{x,y}) \sqcup_{\mathrm{C}(\varkappa''_{x,y})} \mathcal{N} g_1(\top_{\mathrm{C}(\varkappa''_{x,y})}) & \text{if } \varkappa = \varkappa''_{x,y} \\ \mathbf{R}^\sharp(\varkappa) & \text{otherwise} \end{cases}$$

Then, we put

$$[\![x.\mathfrak{r}\sharp\mathfrak{f} := y]\!]^\sharp(\mathbf{A}, \mathbf{R}^\sharp) = \varrho^\sharp_{\mathfrak{M}}(\mathbf{A}, \mathbf{R}^\sharp_\dagger)$$

Note that we have made a conservative approximation on access paths. We cannot do much better, since we only have may-alias information which does not allow us to remove access paths in $\mathbf{A}$. The store-based analysis of [SRW98] is able to handle precisely such cases and to remove access paths in common situations (what is called "strong nullification"). However, this analysis cannot distinguish between elements of recursively defined data structures.

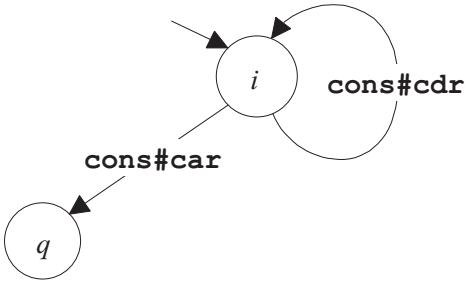
The abstract semantics mimics the concrete one, so that we easily prove the soundness of the previous constructions:

Theorem 14 *The abstract semantics of every instruction* $\mathfrak{i}$ *of the language is sound, i.e. we have:*

$$\forall \mathfrak{m}^\sharp \in \mathfrak{M}^\sharp : \forall \mathfrak{m} \in \gamma^{\mathfrak{M}}(\mathfrak{m}^\sharp) : [\![\mathfrak{i}]\!]\mathfrak{m} \in \gamma^{\mathfrak{M}}([\![i]\!]^\sharp \mathfrak{m}^\sharp)$$

Example 15 *We take Karr's domain of linear equalities and we use the simple widening defined in Sect. 6 to analyze the program described in Figure 1.*

Let $(\mathbf{A}_3, \mathbf{R}_3^\sharp)$ *be the abstract memory configuration obtained at program point 3. We find that* $\mathbf{A}_3(x)$ *and* $\mathbf{A}_3(y)$ *are the same following automaton:*



$\mathbf{R}_3^\sharp(\langle (x,q),(y,q)\rangle)$ *is given by the following system of linear equalities:*

$$\begin{cases} \mathbf{l}.i.\mathtt{cons}\sharp\mathtt{cdr} = \mathbf{r}.i.\mathtt{cons}\sharp\mathtt{cdr} \\ \mathbf{l}.i.\mathtt{cons}\sharp\mathtt{car} = \mathbf{r}.i.\mathtt{cons}\sharp\mathtt{car} = 1 \end{cases}$$

This means that the analysis has been able to infer the exact *set of alias pairs.*

8 Conclusion

We have described an analysis for untyped programs which is able to infer non-uniform aliasing relationships between pointers nested in recursive structures. Our main purpose was to demonstrate that such an analysis could be designed in a simple and modular way. The abstract domain has been built stepwise by successive abstractions of its base components. The abstract semantics of the language has also been specified piecewise. Reusing abstract iteration sequences to construct the abstract closure, for example, allowed us to give a systematic construction of this rather complex operation.

However, this approach is limited by the fact that we enforce an abstract aliasing relation to be *transitively closed*. An aliasing relation abstracts a set of equivalence relations, but the union of such relations is not necessarily transitive. For example, we cannot capture the information that a sorting algorithm does not create aliasing between the elements of the sorted list. In order to handle such cases, we need to modify the abstract semantics and to replace the abstract closure by an operation which describes precisely the new alias pairs created by a destructive assignment. This is done in Deutsch's analysis [Deu92a] for instance, but the design of our semantics would have been much more complicated. We leave this extension as future work.

Acknowledgement

I wish to thank Radhia Cousot, Patrick Cousot, Ian Mackie and the anonymous referee for helpful comments on the first version of this paper.

References

[BW90] M. Barr and C. Wells. *Category Theory for Computing Science*. Prentice Hall, 1990.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, U.S.A., 1977.

[CC79] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proceedings of the Sixth Conference on Principles of Programming Languages POPL'79*. ACM Press, 1979.

[CC92a] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of logic and computation*, 2(4):511–547, August 1992.

[CC92b] P. Cousot and R. Cousot. Comparing the Galois connection and widening-narrowing approaches to abstract interpretation. In *Programming Language Implementation and Logic Programming, Proceedings of the Fourth International Symposium, PLILP'92*, volume 631 of *Lecture Notes in Computer Science*, pages 269–295. Springer-Verlag, 1992.

[CC95a] P. Cousot and R. Cousot. Compositional and inductive semantic definitions in fixpoint, equational, constraint, closure-condition, rule-based and game theoretic form. In *Conference on Computer-Aided Verification, Seventh International Conference, CAV'95*, volume 939 of *Lecture Notes in Computer Science*, pages 293–308. Springer-Verlag, 1995. Invited paper.

[CC95b] P. Cousot and R. Cousot. Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In *Conference Record of FPCA'95*. ACM Press, 1995.

[CH78] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the Fifth Conference on Principles of Programming Languages*. ACM Press, 1978.

[Cou81] P. Cousot. Semantic foundations of program analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 10, pages 303–342. Prentice-Hall, Inc., Englewood Cliffs, 1981.

[Deu92a] A. Deutsch. *Operational models of programming languages and representations of relations on regular languages with application to the static determination of dynamic aliasing properties of data.* PhD thesis, University Paris VI (France), 1992.

[Deu92b] A. Deutsch. A storeless model of aliasing and its abstraction using finite representations of right-regular equivalence relations. In *Proceedings of the 1992 International Conference on Computer Languages*, pages 2–13. IEEE Computer Society Press, Los Alamitos, California, U.S.A., 1992.

[Deu94] A. Deutsch. Interprocedural may-alias analysis for pointers: beyond k-limiting. In *ACM SIGPLAN'94 Conference on Programming Language Design and Implementation.* ACM Press, 1994.

[Eil74] S. Eilenberg. *Automata, Languages and Machines*, volume A. Academic Press, 1974.

[GdW94] J.P. Gallagher and D.A. de Waal. Fast and precise regular approximations of logic programs. In *Proceedings of the International Conference on Logic Programming*, pages 599–613, 1994.

[Gra91] P. Granger. Static analysis of linear congruence equalities among variables of a program. In *TAPSOFT'91*, volume 493. Lecture Notes in Computer Science, 1991.

[GS84] F. Gécseg and M. Steinby. *Tree Automata.* Akademiai Kiado, Budapest, 1984.

[Hei94] N. Heintze. Set-based analysis of ML programs. In *Proceedings of the ACM Conference on Lisp and Functional Programming.* ACM Press, 1994.

[HJ94] N. Heintze and J. Jaffar. Set constraints and set-based analysis. In Alan Borning, editor, *Principles and Practice of Constraint Programming*, volume 874 of *Lecture Notes in Computer Science.* Springer, May 1994.

[Jon81] H.B.M. Jonkers. Abstract storage structures. In De Bakker and Van Vliet, editors, *Algorithmic languages*, pages 321–343. IFIP, 1981.

[Kar76] M. Karr. Affine relationships among variables of a program. *Acta Informatica*, pages 133–151, 1976.

[Kel74] G.M. Kelly. On clubs and doctrines. In A. Dold and B. Eckmann, editors, *Category seminar*, volume 420 of *Lecture Notes in Mathematics*, pages 181–256. Springer Verlag, 1974.

[KS74] G.M. Kelly and R. Street. Review of the elements of 2-categories. In A. Dold and B. Eckmann, editors, *Category seminar*, volume 420 of *Lecture Notes in Mathematics*, pages 75–103. Springer Verlag, 1974.

[Mac71] S. Mac Lane. *Categories for the Working Mathematician.* Graduate Texts in Mathematics. Springer Verlag, 1971.

[SRW98] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *Transactions on Programming Languages and Systems*, 1998.

[VCL94] P. Van Hentenryck, A. Cortesi, and B. Le Charlier. Type Analysis of Prolog using Type Graphs. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation*, SIGPLAN Notices, pages 337–348. Association for Computing Machinery, 1994.

[Ven96] A. Venet. Abstract cofibered domains: Application to the alias analysis of untyped programs. In *Proceedings of the Third International Static Analysis Symposium SAS'96*, volume 1145 of *Lecture Notes in Computer Science*, pages 366–382. Springer-Verlag, 1996.